August 1988

UILU-ENG-88-2249
ACT-99

(2)

COORDINATED SCIENCE LABORATORY
College of Engineering
Applied Computation Theory

INSTRUCTION SETS FOR PARALLEL RANDOM ACCESS MACHINES

Jerry Lee Trahan

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-88-2249 (ACT-99)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Ave. Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) 800 N. Quincy St. Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Joint Services Electronics Program 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-84-C-0149
8c. ADDRESS (City, State, and ZIP Code) 800 N. Quincy St. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

Instruction Sets for Parallel Random Access Machines

12. PERSONAL AUTHOR(S)
Trahan, Jerry Lee

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	August, 1988	169

16 SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	parallel random access machine, parallel computation, computational complexity, instruction set, multiplication, division, shift

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

In this report, we compare the computational power of time bounded Parallel Random Access Machines (PRAMs) with different instruction sets. A basic PRAM can perform the following operations in unit-time: addition, subtraction, Boolean operations, comparisons, and indirect addressing. Multiple processors may concurrently read and concurrently write a single cell. Let PRAM[*op*] denote the class of PRAMs with the basic instruction set augmented with the set *op* of instructions. Let ↑ and ↓ denote unrestricted left and right shift, respectively.

(over

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. ABSTRACT (continued)

We prove that polynomial time on a PRAM[$*$] or on a PRAM[$*,\div$] or on a PRAM[$\uparrow,\downarrow$] is equivalent to polynomial space on a Turing machine (PSPACE). This extends the result that polynomial time on a basic PRAM is equivalent to PSPACE (Fortune and Wyllie, 1978) to hold when the PRAM is allowed unit-time multiplication or division or unrestricted shifts. It also extends to the PRAM the results that polynomial time on a random access machine (RAM) with multiplication is equivalent to PSPACE (Hartmanis and Simon, 1974) and that polynomial time on a RAM with shifts (that is, a vector machine) is equivalent to PSPACE (Pratt and Stockmeyer, 1976; Simon, 1977).

This report establishes that the class of languages accepted in polynomial time on a PRAM[$*,\uparrow,\downarrow$] contains the class of languages accepted in exponential time on a nondeterministic Turing machine (NEXPTIME) and is contained in the class of languages accepted in exponential space on a Turing machine. This result is notable because if, as has been conjectured, NEXPTIME properly contains PSPACE, then a PRAM[$*,\uparrow,\downarrow$] is more powerful, to within a polynomial factor in time, than a PRAM with one of the other instruction sets.

We present efficient simulations of PRAMs with enhanced instruction sets by sequential RAMs with the same instruction sets. This report presents simulations of probabilistic PRAMs by deterministic PRAMs, using parallelism to replace randomness. We also give simulations of PRAM[*op*]s by PRAMs, where both the simulated machine and the simulating machine are exclusive read, exclusive write machines.

INSTRUCTION SETS FOR PARALLEL RANDOM ACCESS MACHINES

BY

JERRY LEE TRAHAN

B.S., Louisiana State University and A. & M. College, 1983
M.S., University of Illinois, 1986

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1988

Urbana, Illinois

Abstract

In this thesis, we compare the computational power of time bounded Parallel Random Access Machines (PRAMs) with different instruction sets. A basic PRAM can perform the following operations in unit-time: addition, subtraction, Boolean operations, comparisons, and indirect addressing. Multiple processors may concurrently read and concurrently write a single cell. Let PRAM[*op*] denote the class of PRAMs with the basic instruction set augmented with the set *op* of instructions. Let $\uparrow$ and $\downarrow$ denote unrestricted left and right shift, respectively.

We prove that polynomial time on a PRAM[*] or on a PRAM[*,÷] or on a PRAM[$\uparrow$,$\downarrow$] is equivalent to polynomial space on a Turing machine (*PSPACE*). This extends the result that polynomial time on a basic PRAM is equivalent to *PSPACE* (Fortune and Wyllie, 1978) to hold when the PRAM is allowed unit-time multiplication or division or unrestricted shifts. It also extends to the PRAM the results that polynomial time on a random access machine (RAM) with multiplication is equivalent to *PSPACE* (Hartmanis and Simon, 1974) and that polynomial time on a RAM with shifts (that is, a vector machine) is equivalent to *PSPACE* (Pratt and Stockmeyer, 1976; Simon, 1977).

This thesis establishes that the class of languages accepted in polynomial time on a PRAM[*,$\uparrow$,$\downarrow$] contains the class of languages accepted in exponential time on a nondeterministic Turing machine (*NEXPTIME*) and is contained in the class of languages accepted in exponential space on a Turing machine. This result is notable because if, as has been conjectured, *NEXPTIME* properly contains *PSPACE*, then a PRAM[*,$\uparrow$,$\downarrow$] is more powerful, to within a polynomial factor in time, than a PRAM with one of the other instruction sets.

We present efficient simulations of PRAMs with enhanced instruction sets by sequential RAMs with the same instruction sets. This thesis presents simulations of probabilistic PRAMs by deterministic PRAMs, using parallelism to replace randomness. We also give simulations of PRAM[*op*]s by PRAMs, where both the simulated machine and the simulating machine are exclusive read, exclusive write machines.

Acknowledgments

I would like to express my great appreciation to Professor Michael Loui for his patient and skillful teaching and his encouragement. I would like to thank Professor Vijaya Ramachandran for her valuable insights and suggestions.

I wish to thank my wife [redacted] for her support, patience, and confidence. I also want to thank my parents for their constant support throughout my education.

This research was supported by the Joint Services Electronics Program (U. S. Army, U. S. Navy, U. S. Air Force) under Contract N00014-84-C-0149.

Table of Contents

List of Figures

Chapter 1. Introduction

An important model of parallel computation is the *Parallel Random Access Machine* (*PRAM*), which comprises multiple processors that execute instructions synchronously and share a common memory. Formalized by Fortune and Wyllie (1978) and Goldschlager (1982), the PRAM is a much more natural model of parallel computation than older models such as combinational circuits and alternating Turing machines (Ruzzo, 1981) because the PRAM abstracts the salient features of a modern multiprocessor computer. Eventually an algorithm developed for the PRAM can be implemented on a parallel network computer such as a mesh-connected array computer (Thompson and Kung, 1977), a hypercube machine (Seitz, 1985), a cube-connected cycles machine (Preparata and Vuillemin, 1981), or a bounded degree processor network (Alt *et al.*, 1987); on all network computers the routing of data complicates the implementation of algorithms.

A number of shared memory machines have been built, such as the Cedar (Kuck, 1986), Cray X-MP (Chen, 1983), NYU Ultracomputer (Schwartz, 1980; Gottlieb *et al.*, 1983), and RP3 (Pfister *et al.*, 1985).

The PRAM provides the foundation for the design of highly parallel algorithms (Luby, 1986; Miller and Reif, 1985; among many others). This model permits the exposure of the intrinsic parallelism in a computational problem because it simplifies the communication of data through a shared memory.

Because of the widespread use of the PRAM model, further advances in research on parallel computation demand a thorough understanding of its capabilities. In particular, we study the effect of the instruction set on the performance of the PRAM.

To quantify differences in computational performance, we determine the time complexities of simulations between PRAMs with different instruction sets. We focus on the computational complexity of simulations between PRAMs with the following operations:

multiplication
division
arbitrary left shift
arbitrary right shift
probabilistic choice

We prove that polynomial time on PRAMs with unit-time multiplication and division or on PRAMs with unit-time ur restricted shifts is equivalent to polynomial space on Turing machines (TMs). Consequently, PRAMs with unit-time multiplication and division and PRAMs with unit-time unrestricted shifts are at most polynomially faster than the standard PRAMs, which do not have these powerful instructions. These results are surprising for two reasons. First, for a sequential random access machine (RAM), adding unit-time multiplication (*) or unit-time unrestricted left shift ($\uparrow$) seems to increase its power:

$RAM\text{-}PTIME = PTIME$ (Cook and Reckhow, 1973),
$RAM[*]\text{-}PTIME = PSPACE$ (Hartmanis and Simon, 1974),
$RAM[\uparrow]\text{-}PTIME = PSPACE$ (Simon, 1977),

whereas adding one of these operations to a PRAM does *not* increase its power by more than a polynomial in time. Second, despite the potential speed offered by massive parallelism, a sequential RAM with unit-cost multiplication or unrestricted shifts is just as powerful, within a polynomial amount of time, as a PRAM with the same additional operation.

The basic PRAM has unit-cost addition, subtraction, Boolean operations, comparisons, and indirect addressing. Let PRAM[*op*] denote the class of PRAMs with the basic instruction set augmented with the set *op* of instructions. Let $PRAM[op]\text{-}TIME(T(n))$ denote the class of languages recognized by PRAM[*op*]s in time $O(T(n))$ on inputs of length

n, $PRAM[op]$–$PTIME$ the union of $PRAM[op]$–$TIME(T(n))$ over all polynomials $T(n)$, and $PRAM[op]$–$POLYLOGTIME$ the union of $PRAM[op]$–$TIME(T(n))$ over all $T(n)$ that are polynomials in $\log n$.

We establish the following new facts about PRAMs. Recall that $PSPACE$ = $PRAM$–$PTIME$ (Fortune and Wyllie, 1978).

$$\begin{aligned} PSPACE &= PRAM[*]\text{–}PTIME \\ &= PRAM[*,+]\text{–}PTIME \\ &= PRAM[+]\text{–}PTIME \\ &= PRAM[\uparrow,\downarrow]\text{–}PTIME \end{aligned} \tag{1}$$

$$\begin{aligned} PRAM\text{–}POLYLOGTIME &= PRAM[*]\text{–}POLYLOGTIME \\ &= PRAM[*,+]\text{–}POLYLOGTIME \\ &= PRAM[+]\text{–}POLYLOGTIME \\ &= PRAM[\uparrow,\downarrow]\text{–}POLYLOGTIME \end{aligned} \tag{2}$$

$$NEXPTIME \subseteq PRAM[*,\uparrow]\text{–}PTIME \subseteq EXPSPACE \tag{3}$$

$$PRAM[*]\text{–}PTIME = RAM[*]\text{–}PTIME \tag{4}$$

$$PRAM[\uparrow,\downarrow]\text{–}PTIME = RAM[\uparrow,\downarrow]\text{–}PTIME \tag{5}$$

These facts follow from theorems in Chapters 4-8, which give more precise time and space bounds.

Chandra and Stockmeyer (1976) and Goldschlager (1978) put forward the *Parallel Computation Thesis:* time on a "reasonable" parallel machine is polynomially related to space on a logarithmic-cost sequential machine (for example, a TM). For a thorough discussion of restrictions necessary for a "reasonable" parallel machine, see Parberry (1987). Basically, a parallel machine is reasonable if the number of processors is restricted to an exponential and the length of cell contents is bounded by an exponential.

The results in (1) are the parallel analogues of the results of Hartmanis and Simon (1974) and Simon (1977) for sequential RAMs. Because of the very long numbers that the RAM[*] and RAM[↑,↓] can generate and because of the equivalence of polynomial time on these models to *PSPACE*, the RAM[*] and RAM[↑,↓] have sometimes been viewed as "parallel." Thus, the PRAM[*] and PRAM[↑,↓] may be viewed as "doubly parallel." The results in (1) are therefore also significant in that introducing unbridled parallelism to a random access machine with unit-time multiplication or unit-time unrestricted shift decreases the running time by at most a polynomial amount.

The results in (2) are notable because of their possible implications for the robust class *NC*, which can be characterized by several different models of parallel computation (Cook, 1985). If we could reduce the number of processors used by the simulation of a PRAM[*], PRAM[*,+], or PRAM[↑,↓] by a PRAM from an exponential number to a polynomial number, then *NC* would be the languages accepted by PRAM[*]s, PRAM[*,+]s, or PRAM[↑,↓]s, respectively, in polylog time with a polynomial number of processors.

Simon (1981a) showed that $PSPACE \subseteq RAM[*, \uparrow]\text{-}PTIME \subseteq EXPSPACE$. The results in (3) show that the same upper bound holds for a parallel RAM[*,↑] and give a lower bound stronger than *PSPACE*, separating a *PRAM* [*, ↑] from PRAMs with the instruction sets previously considered, since it is widely believed that *NEXPTIME* strictly includes *PSPACE*. This result does not contradict the Parallel Computation Thesis, however, since the numbers created by multiplication and shift together are too long and complex to be "reasonable."

The results in (4) and (5) are implied by (1), but we note these because we strengthened the time bounds by more direct simulations than through the results in (1).

We have also proved some results for probabilistic PRAMs (*prob*-PRAMs). Reif (1984) simulated a *prob*-RAM[*,÷] M with time bound $T(n)$, memory bound $S(n)$, and integer bound $I(n)$ by a *prob*-PRAM[*,÷] in time $O(S(n) \log I(n) + \log T(n))$. We simulate M by a *deterministic* PRAM[*,÷] in time $O(S(n) \log I(n) \log (S(n)T(n)))$, then extend the simulation to a *prob*-PRAM[*,÷].

In Chapter 2, we review the relevant literature, and in Chapter 3, we formally define our model. Chapters 4 and 5 contain the multiplication and division results. Chapter 6 contains our theorems relating to PRAMs with shifts. We establish bounds on the computational power of time-bounded PRAMs with both multiplication and shift in Chapter 7. Chapter 8 contains our work on PRAMs with probabilistic choice. Chapter 9 contains simulations of PRAMs with enhanced instruction sets by sequential RAMs with the same instruction sets. In Chapter 10, we discuss the effects of variations in the definition of the basic PRAM on our simulations, and in Chapter 11, we summarize our results and present some open problems arising from our work.

A preliminary version of Chapters 4, 5, and 6 appeared at the 22nd Annual Conference on Information Sciences and Systems in Princeton, New Jersey, in March 1988 (Trahan *et al.*, 1988).

Chapter 2. Literature Review

In this chapter, we survey research done on instruction sets for RAMs and PRAMs. We also review results relating the PRAM to other models of parallel computation and briefly discuss previous work on probabilistic models of computation. Unless otherwise specified, a RAM has unit-cost addition, subtraction, Boolean operations on bit vectors, conditional jumps, and indirect reads and writes.

• *Instruction sets*

Hartmanis (1971) introduced the Random Access Stored Program (RASP) machine, the first computational model with random access memory. Cook and Reckhow (1973) presented a restricted RAM model whose instruction set did not include Boolean operations. We shall call this model an *rRAM*. A RASP and rRAM can simulate each other with at most a constant factor loss in time. Let $l(y)$ denote the execution time of an instruction on an rRAM, where y is the size of the operands. They simulated a Turing machine (TM) running in time $T(n)$ by an rRAM running in time $O(T(n) \cdot l(T(n)))$ and an rRAM running in time $T(n)$ by a TM running in time $O(T^3(n))$, if $l(y)$ is constant, or $O(T^2(n))$, if $l(y)$ is logarithmic. They also established a strict time hierarchy for rRAMs. Wiedermann (1983) improved the bound for the logarithmic time measure to $O(T^2(n) / \log US(n))$, where $US(n)$ is the number of registers used by the rRAM.

Schönhage (1980) proved that for a *successor RAM*, that is, a RAM without Boolean operations and whose set of arithmetic instructions is restricted to adding one to a register's contents, $successor\text{-}RAM\text{-}PTIME = P$.

The four papers upon which our work is squarely based are Hartmanis and Simon (1974), Pratt and Stockmeyer (1976), Simon (1977), and Fortune and Wyllie (1978).

Hartmanis and Simon studied the RAM[*], proving $RAM[*]\text{-}PTIME = PSPACE$. Thus, the inclusion of multiplication strengthens the RAM[*] over the RAM of Cook and Reckhow. To simulate a RAM[*] on a Turing machine, Hartmanis and Simon treated the long numbers generated with multiplication by manipulating only individual bits of registers. In only polynomial space, a TM can address any bit of a register whose contents can grow exponentially long. They also established that the same results hold if the multiplication instruction is replaced by an instruction for the concatenation of two strings.

Pratt and Stockmeyer studied a restricted RAM[$\uparrow$,$\downarrow$], or *vector machine*, as they called it. A vector machine does not have arithmetic operations, and shift distances are restricted to a polynomial, hence restricting the lengths of register contents to an exponential. They proved that the class of languages recognized in polynomial time on a vector machine is equal to *PSPACE*, also by manipulating individual bits of registers. A vector machine is often viewed as a parallel computer in which each processor handles a single bit: a Boolean operation is seen as an instruction executed simultaneously by each processor and a shift is seen as a communication step between processors. Simon (1977) removed the restrictions on shift distances and allowed addition and subtraction, showing that the class of languages recognized in polynomial time on this machine (a RAM[$\uparrow$,$\downarrow$] in our notation) is still equal to *PSPACE*. He dealt with the extremely long numbers that a RAM[$\uparrow$,$\downarrow$] can generate by working with encodings of register contents. We discuss this encoding in detail in Chapter 6.

Fortune and Wyllie (1978) introduced the PRAM model, establishing that the class of languages recognized in polynomial time on this model is also equal to *PSPACE*. They showed that in space $O(T^2(n))$, a TM can simulate a PRAM running in time $T(n)$. The TM executes a procedure that checks that at time t, processor P_j executed instruction number i,

leaving c in its accumulator. This procedure is recursive in time. Fortune and Wyllie also showed that for nondeterministic PRAMs, the class of languages recognized in polynomial time is equal to the class of languages recognized in exponential time by a nondeterministic TM.

Hartmanis and Simon (1974), Pratt and Stockmeyer (1976), and Fortune and Wyllie (1978) all used basically the same method to simulate a space-bounded TM. For all pairs of TM configurations α and β, the simulating machine Q first generates a transition matrix indicating whether the simulated TM can make a transition from α to β in one step. By successive squarings, Q then computes the transitive closure of the matrix, which gives the $T(n)$-step transition matrix, and reads from the resulting matrix whether the TM starting in the initial configuration reaches an accepting configuration in $T(n)$ steps. Fortune and Wyllie assigned one processor to each configuration; Pratt and Stockmeyer put the entire transition matrix into a single register, then squared it with shifts and Boolean operations; and Hartmanis and Simon did the same as Pratt and Stockmeyer, using multiplication as a restricted shift operator. It is interesting to note that Hartmanis and Simon used only the shifting ability of multiplication, but no other property.

Recall that a vector machine is a RAM[$\uparrow$,$\downarrow$] without addition or subtraction and whose register content lengths are bounded by an exponential. Stockmeyer (1976) established a separation between time-bounded vector machines and time-bounded RAM[*]s *without Boolean operations*. He described a language that can be accepted in constant time by a vector machine, but requires linear time on a RAM[*] without Boolean operations.

Let *RP* denote the class of languages accepted in polynomial time on a probabilistic Turing machine. (We discuss *RP* later in this chapter.) In some early work focusing on

powerful instruction sets, Schönhage (1979) demonstrated that for a RAM without Boolean operations, $RAM[*]\text{-}PTIME \subseteq RP$ and $NP \subseteq RAM[*, +]\text{-}PTIME$. Thus, the exclusion of Boolean operations appears to weaken the RAM[*] model since $RP \subseteq PSPACE$.

Two parallel models without shared memory were presented by Goldschlager (1982) and Savitch and Stimson (1979). Goldschlager called his model a conglomerate. A *conglomerate* is an infinite collection of identical finite-state machines connected according to some connection function. The computing ability of a conglomerate arises from the connection function. Goldschlager argued that the set of feasibly buildable machines is the set of conglomerates whose connection functions can be computed in polynomial space on a TM. Savitch and Stimson called their model a PRAM; let us call it a *tree*-PRAM to distinguish it from our version. The *tree*-PRAM differs from our PRAM in the communication between processors. In the PRAM, any processor may communicate with any other through a shared memory. In the *tree*-PRAM, there is no shared memory; a processor may communicate only with its parent (the processor that activated it) and its children (the processors that it activates). Savitch and Stimson proved the equivalence of polynomial time on a *tree*-PRAM and polynomial space on a RAM, where space on a RAM is defined as the sum of the lengths of the contents of the registers at any time.

Savitch (1982) considered the *tree*-PRAM with an expanded instruction set, allowing the processors three string manipulation instructions: concatenation of two strings, extraction of the first half of a string, and extraction of the second half of a string. He proved that the class of languages accepted in polynomial time by this model equals *PSPACE*. Hartmanis and Simon (1974) proved that the class of languages recognized in polynomial time on a sequential RAM with concatenation is equal to *PSPACE*, so the parallelism

allowed in the *tree*-PRAM with the same instruction set provides no more power, to within a polynomial in time.

Division is a natural instruction for us to consider as a part of our instruction set. Hartmanis and Simon (1974) proved that $RAM[*, +]\text{–}PTIME = PSPACE$. Simon (1981a) showed that a RAM with division and left shift can be extremely powerful. He proved $RAM[\uparrow, \div]\text{–}PTIME = ER$, where ER is the class of languages accepted in time

$$2^{2^{2^{\cdot^{\cdot^{2}}}}} \Big\} n \uparrow$$

by Turing machines. At first glance, this is a surprising result: for a RAM with left shift and right shift, we already know that $RAM[\uparrow, \downarrow]\text{–}PTIME = PSPACE$. Simon proved $ER \subseteq RAM[\uparrow, +]\text{–}PTIME$ by building very long integers with the left shift operation and then manipulating them as both integers and binary strings. Division is used to generate a complex set of strings representing all possible TM configurations. (Note that right shift cannot replace division in building these strings.)

In the same paper, Simon studied the inclusion of both multiplication and shift in the instruction set and a probabilistic version of a RAM. Using the same encoding that he had previously used to simulate a RAM[$\uparrow, \downarrow$], he proved $RAM[*, \uparrow]\text{–}PTIME \subseteq EXPSPACE$. A RAM with multiplication and shift can generate more complex numbers than a RAM with shift alone; hence, the size of the encoding of the numbers increases. This complexity accounts for the increase from *PSPACE* to *EXPSPACE*. Simon called his probabilistic model a *stochastic RAM*. This RAM has a random number generator that, on operand x, returns a random integer uniformly distributed in the interval $[0, x]$. A stochastic RAM accepts input ω if the probability of reaching an accepting state is greater than ½. Simon exploited this acceptance condition in his proofs, proving the following results:

stochastic –RAM [↑]*–PTIME = ER*, *NEXPTIME* ⊆ *stochastic –RAM* [*]*–PTIME*, and *NEXPTIME* ⊆ *stochastic –PRAM –PTIME*. Simon claimed that this model is equivalent to a *prob*-PRAM where each processor has an unbiased coin of its own (that is, if the stochastic RAM has sufficiently powerful arithmetic instructions (* or ↑) to generate long numbers quickly enough).

• *Relationships between RAMs and TMs*

Hopcroft *et al.* (1975) described a simulation of a TM running in time $T(n) \geq n \log n$ by a RAM running in time $O(T(n) / \log T(n))$. The key to the simulation is that, for each block of TM tape of size $O(\log T(n))$, the RAM precomputes a look-up table containing the contents of that block after $O(\log T(n))$ steps of the TM.

Katajainen *et al.* (1988) proved that a $T(n)$ time-bounded, $S(n)$ space-bounded, and $U(n)$ output-length-bounded TM can be simulated by a RAM in $O(T(n) + (n+U(n)) \operatorname{loglog} S(n))$ time.

Measuring space on a RAM as the sum of the lengths of the contents of all registers used by the RAM, Slot and van Emde Boas (1988) established that an off-line TM running in space $S(n)$ can simulate an off-line RAM running in space $S(n)$. They also showed that a simulation with no loss in space is not possible for the on-line versions.

• *Relationships between PRAMs and other computational models*

Stockmeyer and Vishkin (1984) established that parallel time and number of processors on a concurrent read, concurrent write (CRCW) PRAM correspond to depth and size for unbounded fan-in circuits. Time and depth correspond to within a constant factor; number of processors and size correspond to within a polynomial. Because we will use their results frequently, we state their theorems here.

Theorem 2.1. (Stockmeyer and Vishkin, 1984) Let Z be a PRAM with time bound $T(n)$ and processor bound $P(n)$. There is an unbounded fan-in circuit C_n that simulates Z in depth $O(T(n))$ and size $O(P(n)T(n)[T(n)(n+T(n)) + (n+T(n))^3 + (n+T(n))(n+P(n)T(n))])$.

Proof (sketch). See the proof of Lemma 9.3.1.1 for a description of the circuit C_n. □

Theorem 2.2. (Stockmeyer and Vishkin, 1984) Let C be an unbounded fan-in circuit of size S and depth T with n inputs. There is a PRAM Z that simulates C in time $O(T)$ with $O(S + n)$ processors.

Proof (sketch). Each processor of Z simulates a wire in C, and each memory cell in Z simulates a gate in C. Machine Z simulates C one level at a time. To simulate an OR gate, the corresponding cell is initially set to 0. A processor that is simulating a wire into that gate writes 1 into the cell if its wire carries a 1, and it does not write if its wire carries a 0. To simulate an AND gate, the corresponding cell is initially set to 1. A processor that is simulating a wire into that gate writes 0 into the cell if its wire carries a 0, and it does not write if its wire carries a 1. □

Stockmeyer and Vishkin also related time and number of processors on a CRCW PRAM to number of alternations and space on an alternating TM.

The *Hardware Modification Machine (HMM)* was defined by Dymond and Cook (1980). An HMM comprises a set of finite-state machines, called *units*. Each unit reads a constant number of input symbols from neighboring units and computes an output symbol based on its inputs and current state at each time step. Each unit has "taps" on the outputs of other units through which it reads its inputs. At each step, a unit may relocate one of its taps. Ruzzo (1985) defined a restricted PRAM (rPRAM) with the following constraints: each processor has a finite local memory; there are no Boolean operations; the only

arithmetic instructions are successor and doubling; and shared memory is divided into blocks, where processor P_i owns the ith block. Ruzzo showed that HMMs and rPRAMs are equivalent, to within a constant factor, in both time and hardware simultaneously.

Dymond and Tompa (1985) established that for all $T(n) \geq n$, $DTIME(T(n)) \subseteq PRAM\text{-}TIME(T^{1/2}(n))$. Their simulation features the use of a look-up table like that of Hopcroft *et al.* (1975).

Hong (1986) gave the following informal definitions of space, parallel time, and sequential time as they apply to various models.

> (1) The space (width) is the maximum length of intermediate information in the computation.
>
> (2) The parallel time (reversal) is the total number of phases. A phase is a period of the computation during which no information written on the work space is read during the same period.
>
> (3) The sequential time is the total number of primitive operations during the computation.

He related these complexity measures in the Similarity Principle: "All idealized computational models are similar in the sense that their parallel time, their space, and their sequential time complexities are polynomially related simultaneously." In support of the Similarity Principle, Hong established that the following computational models are similar:

> *TM (reversal, space)*
>
> *RAM*[*,÷] *(reversal, space)*
>
> *Vector Machine (time, space)*
>
> *Uniform Circuit (depth, width)*
>
> *PRAM*[*,÷] *(time, space)*

Parberry (1986) demonstrated that, with enough processors, a CRCW PRAM can compute any recursive function in constant time.

Ranade (1987) presented a simulation of a P processor CRCW PRAM on a P node butterfly network such that the network simulates each step of the PRAM in $O(\log P)$ time with high probability. He used randomness only to select a hash function to distribute the shared memory of the PRAM among the nodes of the butterfly network. Routing in the network is deterministic and oblivious.

Alt *et al.* (1987) gave a nonuniform deterministic simulation of an exclusive read, exclusive write (EREW) PRAM on a bounded degree processor network. If the PRAM has P processors and S cells of shared memory, then the network simulates each step in $O(\log P \log S)$ time. If the PRAM is CRCW, then the simulation time increases to $O(\log^2 P \log S)$. For a restricted network, they proved a lower bound of $\Omega(\log P \log S / \operatorname{loglog} S)$ time to simulate an EREW PRAM.

Cook (1980), Vishkin (1983b), and Parberry (1987) provided good surveys on parallel models of computation. Karp and Ramachandran (1988) gave a good survey of parallel algorithms.

• *Probabilistic models*

To close this chapter, we survey some of the work done on probabilistic models of computation.

de Leeuw *et al.* (1956) showed that the ability to make random choices does not change the (unbounded) computational power of Turing machines (TMs). Santos (1969, 1971) investigated a more general notion of probabilistic Turing machine (PTM) allowing biased random choices; his PTMs have the same power as PTMs allowing only unbiased choices.

Gill (1977) defined a PTM as being allowed to toss an unbiased coin. He defined Blum complexity measures for time and space. *PP* is defined as the class of languages recognized by polynomial time bounded PTMs (no error bounds). *BPP* is the class of languages recognized by polynomial time bounded PTMs with bounded error probability. *VPP* is the class of languages recognized by polynomial time bounded PTMs with zero error probability for inputs not in the language. (Note: *VPP* is more often designated *R* or *RP*; I will henceforth use *RP* to designate this class.) *ZPP* is the class of languages recognized by PTMs with polynomial bounded average time and zero error probability ($RP \cap co\text{-}RP$). Gill showed the following:

$$\subseteq BPP \subseteq$$

$$P \subseteq ZPP \subseteq RP \qquad PP \subseteq PSPACE.$$

$$\subseteq NP \subseteq$$

He also showed that only partial recursive functions are probabilistically computable. He described a palindrome-like language that can be recognized by a fixed one-tape PTM faster than by any one-tape deterministic TM (DTM). He proved that a PTM with time bound $T(n)$ can be simulated by a DTM in time $O(T^2(n)2^{T(n)})$.

Simon (1981b) proved *RPSPACE* = *PSPACE* (where *RPSPACE* is polynomial space on a PTM). He accomplished this by showing *RPSPACE* = *RAM*[*]–*PTIME*; the RAM[*] treats the computation of the PTM as a Markov chain to compute the probability of acceptance.

Rabin (1976) gave a general discussion of probabilistic algorithms, considering three types: (1) algorithms that halt within expected time $f(n)$, always producing the correct

answer and always terminating (sometimes called Las Vegas algorithms), (2) algorithms that may produce an erroneous answer, and (3) algorithms that may produce an erroneous answer and may (with probability 0) not halt. (Algorithms of types (2) and (3) are sometimes called Monte Carlo algorithms.) The randomness he allowed is choosing an integer in $\{1, ..., n\}$.

Welsh (1983) surveyed various randomized algorithms. He also surveyed knowledge about *RP* and random log-space (*RL*: $L \subseteq RL \subseteq NL \subseteq P$).

Reif (1984) investigated *prob*-PRAM[*,+] simulations of the *prob*-RAM[*,+]. He used the CREW version. We present his results in detail in Chapter 8. Reif also showed how probabilistic choice can be removed from *prob*-PRAMs with two-sided error by introducing nonuniformity, with some increase in time and processor bounds.

Robson (1984) demonstrated that a *prob*-RAM can simulate a deterministic one-tape TM in expected time $O(T(n) / \text{loglog } T(n))$ under the log-cost criterion. The *prob*-RAM handles the TM tape in blocks. Note that Hopcroft *et al.* (1975) showed a simulation of a multitape TM in $O(T(n) / \log T(n))$ unit-cost time on a deterministic RAM.

Chapter 3. Definitions and Two Key Lemmas

ram *n. 1. A male sheep. 2. (Capital R) A constellation and sign of the zodiac,* Aries. *3. Any of several devices used to drive, batter, or crush by forceful impact.*

pram *n. (Chiefly British) A perambulator: a baby carriage.*

(Morris, 1980)

We study a deterministic PRAM similar to that of Stockmeyer and Vishkin (1984). A PRAM consists of an infinite collection of processors $P_0, P_1, \cdots$, an infinite set of shared memory cells, $c(0), c(1), \cdots$, and a program which is a finite set of instructions labeled consecutively with 1, 2, 3, $\cdots$. All processors execute the same program. Each processor has a program counter. Each processor P_m has an infinite number of local registers: $r_m(0)$, $r_m(1), \cdots$. Each cell $c(j)$, whose *address* is j, contains an integer $con(j)$, and each register $r_m(j)$ contains an integer $rcon_m(j)$.

For convenience we use a PRAM with concurrent read and concurrent write (CRCW) in which the lowest numbered processor succeeds in a write conflict. At time t in a computation of a PRAM, at most 2^t processors are active. Since we are concerned with at least polylog time, there are no significant differences between the concurrent read / concurrent write (CRCW), the concurrent read / exclusive write (CREW), and the exclusive read / exclusive write (EREW) PRAMs because the EREW model can simulate the CRCW model with a penalty of only a logarithmic factor in time (log of the number of processors attempting to simultaneously read or write) (Cook *et al.*, 1986; Borodin and Hopcroft, 1985; Fich *et al.*, 1988b; Vishkin, 1983a). If one or more processors attempt to read a cell at the same time that a processor is attempting to write the same cell, then all reads are performed before the write.

Initially, the input, a nonnegative integer, is in $c(0)$. For all m, register $r_m(0)$ contains m. All other cells and registers contain 0, and only P_0 is active. A PRAM accepts its input if and only if P_0 halts with its program counter on an *ACCEPT* instruction.

In time $O(\log n)$, a processor can compute the smallest n such that $con(0) \leq 2^n - 1$; the PRAM takes this n as the length of the input. Whenever $con(i)$ is interpreted in two's complement representation, we number the bits of $con(i)$ consecutively with 0, 1, 2, $\cdots$, where bit 0 is the rightmost (least significant) bit.

A PRAM Z has time bound $T(n)$ if for all inputs ω of length n, a computation of Z on ω halts in $T(n)$ steps. Z has processor bound $P(\cdot)$ if for all inputs ω of length n, Z activates at most $P(n)$ processors during a computation on ω. We assume that $T(n)$ and $\log P(n)$ are both time-constructible in the simulations of a PRAM[*op*] by a PRAM, so that all processors have values of $T(n)$ and $P(n)$.

We allow indirect addressing of registers and shared memory cells through register contents. The notation $c(r_m(j))$ refers to the cell of shared memory whose address is $rcon_m(j)$, and $r(r_m(j))$ refers to the register of P_m whose address is $rcon_m(j)$.

The basic PRAM model has the following instructions. When executed by processor P_m, an instruction that refers to register $r(i)$ uses $r_m(i)$.

$r(i) \leftarrow k$ (load a constant)
$r(i) \leftarrow r(j)$ (load the contents of another register)
$r(i) \leftarrow c(r(j))$ (indirect read from shared memory)
$c(r(i)) \leftarrow r(j)$ (indirect write to shared memory)
$r(i) \leftarrow r(r(j))$ (indirect read from local memory)
$r(r(i)) \leftarrow r(j)$ (indirect write to local memory)
ACCEPT (halt and accept)
REJECT (halt and reject)
FORK label 1, *label* 2 (P_m halts and activates P_{2m} and P_{2m+1}, setting their program counters to *label* 1 and *label* 2, respectively.)
$r(i) \leftarrow BIT(r(j))$ (read the $rcon_m(j)$th bit of $con(0)$ (the input))
CJUMP $r(j)$ *comp* $r(k)$, *label* (jump to instruction labeled *label* on

condition $rcon_m(j)$ *comp* $rcon_m(k)$, where the
arithmetic comparison *comp* $\in \{<, \leq, =, \geq, >, \neq\}$),
$r(i) \leftarrow r(j)\ O\ r(k)$ for $O \in \{+, -, bool\}$
(addition, subtraction, bitwise Boolean operations)

Processor P_0 can perform a *FORK* operation only once. This restriction is necessary to prevent the activation of multiple processors with identical processor numbers. This is also the reason why P_m halts when it performs a *FORK*.

In the definition of the *FORK* instruction given by Fortune and Wyllie (1978), the processor executing a *FORK* remains active and activates the lowest numbered inactive processor.

For an integer d, define $len(d)$ as the minimum integer w such that $-2^{w-1} \leq d \leq 2^{w-1}-1$. Thus, d has a two's complement representation with w bits. Let $w = \max\{len(rcon_m(j)), len(rcon_m(k))\}$. To perform a Boolean operation on $rcon_m(j)$ and $rcon_m(k)$, the PRAM performs the operation bitwise on the w-bit two's complement representations of $rcon_m(j)$ and $rcon_m(k)$. The PRAM interprets the resulting integer x in w-bit two's complement representation and writes x in $r_m(i)$. We need at least w bits so that the result will correctly be positive or negative.

Let $\uparrow$ (respectively, $\downarrow$) denote the unrestricted left (respectively, right) shift operation: the instruction $r(i) \leftarrow r(j) \uparrow r(k)$ (respectively, $r(i) \leftarrow r(j) \downarrow r(k)$) places $rcon_m(j) \cdot 2^{rcon_m(k)}$ (respectively, $rcon_m(j) \div 2^{rcon_m(k)}$) into $r_m(i)$. The instruction can also be viewed as placing into $r_m(i)$ the result of shifting the binary integer $rcon_m(j)$ to the left (respectively, right) by $rcon_m(k)$ bit positions.

Let *prob*-PRAM denote the class of probabilistic PRAMs in which each processor is allowed to uniformly choose one of a constant size multiset of alternatives at each step.

Instructions have the following form:

$$r(i) \leftarrow r(j) \bigcirc r(k); \alpha_1, \alpha_2, ..., \alpha_g$$

in which $\alpha_1, \alpha_2, ..., \alpha_g$ are instruction labels; the processor executes $r(i) \leftarrow r(j) \bigcirc r(k)$, then uniformly selects one of $\{\alpha_1, \alpha_2, ..., \alpha_g\}$ as the next instruction.

In some variants of the PRAM model, the input is initially located in the first n cells, one bit per cell. We therefore have the instruction "$r(i) \leftarrow BIT(r(j))$" in order to remove the effects of a different input convention. This instruction was also used by Reischuk (1987).

At each step, each active processor simultaneously executes the instruction indicated by its program counter in one unit of time, then increments its program counter by one, unless the instruction causes a jump. On an attempt to read a cell at a negative address, the processor reads the value 0; on an attempt to write a cell at a negative address, the processor does nothing.

The assumption of unit time instruction execution is an essential part of our definition. In a sense, our work is a study of the effects of this unit-cost hypothesis on the computational power of time-bounded PRAMs as the instruction set is varied.

For ease of description, we sometimes allow a PRAM a small constant number of separate memories, which can be interleaved. This allowance entails no loss of generality and only a constant factor time loss.

Lemma 3.1. For all $T(n)$, every language recognized in time $T(n)$ by a PRAM R with q separate shared memories, q a constant, can be recognized in time $O(T(n))$ by a PRAM Z with one shared memory.

Proof. R has q separate shared memories: $mem_0, \cdots, mem_{q-1}$. Let $c_b(k)$ denote the kth cell of mem_b and $con_b(k)$ the contents of that cell.

Z maps $c_b(k)$ of R to $c(qk + b)$. Thus, to simulate an access to $c_b(k)$, Z computes $qk + b$ in constant time in its local memory, then accesses $c(qk + b)$. In this manner, Z simulates each step of R in constant time. □

In Chapter 10, we discuss the effects of variations in the definition of the basic PRAM on our results. In particular, we look at the concurrent read and write capabilities, write conflict resolution scheme, *FORK* operation, and size of local memory.

In the simulations to follow, the simulating PRAM activates *primary* and *secondary* processors. The Activation Lemma tells how the PRAM activates them and how their processor numbers are related.

Activation Lemma. A PRAM R' can activate p primary processors, each with s secondary processors, in $O(\log p + \log s)$ steps.

Proof. In $O(\log p)$ steps, R' activates p primary processors. By definition of the *FORK* command, these processors are numbered $p, p+1, \cdots, 2p-1$. In $O(\log s)$ steps, each of these processors activates s secondary processors. When each primary processor P_g *FORK*s, it sets the program counter of P_{2g} to indicate that it is a primary processor and the program counter of P_{2g+1} to indicate that it is a secondary processor. When each secondary processor P_h *FORK*s, it sets the program counters of P_{2h} and P_{2h+1} to indicate that they are secondary processors. After the $O(\log s)$ steps, the primary and secondary processors are numbered $ps, ps+1, \cdots, 2ps-1$. Processors numbered $js, p \le j \le 2p-1$, are primary processors. Processors numbered $js+k, 0 \le k \le s-1$, are the secondary processors belonging

to P_{js}. Each primary processor is also considered as a secondary processor belonging to itself. Primary processor P_{js} corresponds to processor P_{j-p} of the simulated machine R. □

Let R be a PRAM[*]. By repeated application of the multiplication instruction, R can generate integers of length $O(n2^{T(n)})$ in $T(n)$ steps. By indirect addressing, processors in R can access cells with addresses up to $2^{n2^{T(n)}}$ in $T(n)$ steps, though R can access at most $O(P(n)T(n))$ different cells during its computation. In subsequent chapters, these cell addresses will be too long for the PRAM and TM simulators to write. Therefore, we first construct a PRAM[*] R' that simulates R and uses only short addresses. Similarly, a PRAM[↑,↓] can generate extremely long integers and use them as indirect addresses, so we simulate this by a PRAM[↑,↓] that uses only short addresses.

Associative Memory Lemma. Let $op \subseteq \{*, \uparrow, \downarrow\}$. For all $T(n)$ and $P(n)$, every language recognized with $P(n)$ processors in time $T(n)$ by a PRAM[op] R can be recognized in time $O(T(n))$ by a PRAM[op] R' that uses $O(P^2(n)T(n))$ processors and accesses only cells with addresses in $0, \cdots, O(P(n)T(n))$.

Proof. Let R be an arbitrary PRAM[op] with time bound $T(n)$ and processor bound $P(n)$. We construct a PRAM[op] R' that simulates R in time $O(T(n))$ with $P^2(n)T(n)$ processors, but accesses only cells with addresses in $0,\ldots,O(P(n)T(n))$. R' employs seven separate shared memories: $mem_1, \cdots, mem_7$. R' uses memories mem_1 and mem_2 to simulate the shared memory of R; memories mem_3, mem_4, and mem_5 to simulate the local registers of R; and memories mem_6 and mem_7 for communication among the processors. Let $c_b(k)$ denote the kth cell of mem_b and $con_b(k)$ the contents of that cell. R' organizes the cells of mem_1 and mem_2 in pairs to simulate the memory of R: the first component, $c_1(k)$, holds the address of a cell in R; the second component, $c_2(k)$, holds the contents of that cell. Actually, in order to distinguish address 0 from an unused cell, $c_1(k)$ holds one plus the address. Let

pair (k) denote the kth memory pair. R' organizes the cells of mem_3, mem_4, and mem_5 in triples to simulate the local registers of R: the first component, $c_3(k)$, holds the processor number; the second component, $c_4(k)$, holds the address of a register in R; the third component, $c_5(k)$, holds the contents of that register. Let *triple* (k) denote the kth memory triple. Since R can access at most $O(P(n)T(n))$ cells in $T(n)$ steps, R' can simulate the cells used by R with $O(P(n)T(n))$ memory pairs and triples.

Let P_m denote processor number m of R; let P'_m denote processor number m of R'.

We now describe the operation of R'. In $O(\log P(n))$ steps, R' activates $P(n)$ processors, called *primary processors*. In the next $\log(P(n)T(n))$ steps, each primary processor activates $P(n)T(n)$ *secondary processors*, each of which corresponds to a memory pair and a memory triple.

By the Activation Lemma, primary processor P'_m corresponds to the processor of R numbered $m / P(n)T(n)$. The processors numbered $m + k$, for all k, $0 \le k \le P(n)T(n)-1$, will be the secondary processors belonging to primary processor P'_m. So, if $i < m$, then all secondary processors belonging to P'_i are numbered lower than all secondary processors belonging to P'_m. We exploit this ordering to handle concurrent writes by processors in R.

When the secondary processors of P'_m are not otherwise occupied, they concurrently read $c_6(m)$ at each time step, waiting for a signal from P'_m to indicate their next tasks. R' applies its concurrent read capabilities in this way to implement a constant time broadcast from a primary processor to all of its secondary processors.

Each secondary processor assigns itself to the memory pairs and triples as follows. Each secondary processor P'_j belonging to P'_m, $j = m + k$, handles *pair* (k) and *triple* (k). We call k the *assignment number* of P'_j. P'_j computes its assignment number in constant time.

Suppose R' is simulating step t of R in which P_g writes v in $c(f)$. Then the corresponding primary processor P'_m of R' writes $f+1$ in $c_1(P(n)(T(n)-t)+g+1)$ and v in $c_2(P(n)(T(n)-t)+g+1)$. That is, at step t of R, all primary processors of R' write only cells with addresses in $P(n)(T(n)-t)+1, \cdots, P(n)(T(n)-t+1)$ with the lowest numbered primary processor writing in the lowest numbered cell in the block. The memory holds a copy for every time a processor attempts to write $c(f)$. Figure 3.1 is an example, for $P(n) = 4$ and $T(n) = 3$, showing which primary processors write in which cells of mem_1 and mem_2 at each step of R and the assignment numbers of the secondary processors assigned to those cells. By this ordering, the copy of a cell in R with the current contents (most recently written by lowest numbered processor) is handled by the lowest numbered secondary processor. If at some later step a primary processor P'_m desires to read $con(f)$ of R, then its secondary processors read all copies of $con(f)$ and concurrently write their values in $c_7(m)$. By the write priority rules, the secondary processor reading the current value of $con(f)$ will succeed in the write.

pair # in R'	proc. # in R	assignment #	step of R
1	P_0	1	$t = 3$
2	P_1	2	"
3	P_2	3	"
4	P_3	4	"
5	P_0	5	$t = 2$
6	P_1	6	"
7	P_2	7	"
8	P_3	8	"
9	P_0	9	$t = 1$
10	P_1	10	"
11	P_2	11	"
12	P_3	12	"

Figure 3.1. Memory allocation: $T(n) = 3, P(n) = 4$

Similarly, suppose R' is simulating a step of R in which P_g writes v in $r_g(f)$. Then P'_m writes g in $c_3(P(n)(T(n)-t)+g+1)$, $f+1$ in $c_4(P(n)(T(n)-t)+g+1)$, and v in $c_5(P(n)(T(n)-t)+g+1)$. If at some later step P'_m desires to read $rcon_g(f)$, then its secondary processors read all copies of $rcon_g(f)$ and concurrently write their values in $c_7(m)$.

When a processor P_g of R executes an instruction $r(i) \leftarrow r(j)\ \bigcirc\ r(k)$, it reads $rcon_g(j)$ and $rcon_g(k)$, computes $v := rcon_g(j)\ \bigcirc\ rcon_g(k)$, and writes v in $r_g(i)$. The corresponding processor P'_m of R' simulates this step as follows. If j is negative, then P'_m writes 0 in $c_6(m)$ and $c_7(m)$. Otherwise, P'_m writes g in $c_7(m)$ to indicate that it wishes to read the contents of a register of P_g and writes 0 in $c_6(m)$ to clear it. Each secondary processor of P'_m reads $c_7(m)$ and compares $con_7(m)$ with the value of the first component of its assigned memory triple. P'_m writes $j+1$ in $c_7(m)$ to specify the address of the register it wishes to read. Each secondary processor of P'_m that matches g reads $c_7(m)$ and compares $con_7(m)$ with the second component of its assigned memory triple. If the contents match for secondary processor P'_j, which is assigned *triple*(k), then P'_j writes $con_5(k)$ in $c_7(m)$ and 1 in $c_6(m)$ to inform P'_m that it has found the desired register contents. P'_m reads $c_6(m)$. If $con_6(m) = 0$, then no secondary processor matched the address; hence, $c(j)$ is a cell of R that has not yet been written, and P'_m writes 0 in $r_m(1)$. If $con_6(m) = 1$, then some secondary processor matched the address, and P'_m copies $con_7(m)$ to $r_m(1)$. Next, P'_m writes 0 in $c_6(m)$ and $c_7(m)$ and repeats the process for k, except that P'_m writes $r_m(2)$ instead of $r_m(1)$. (Note: Handling indirect addresses requires two cycles through the above steps.) P'_m then computes $v := rcon_m(1)\ \bigcirc\ rcon_m(2)$, writing v in $r_m(1)$. Next, if i is negative, then P'_m does nothing. Otherwise, suppose R' is simulating step t of R. Each primary processor keeps track of t in its local memory. Then P'_m writes g in $c_3(P(n)(T(n)-t)+g+1)$, $i+1$ in

$c_4(P(n)(T(n)-t)+g+1)$, and v in $c_5(P(n)(T(n)-t)+g+1)$ to complete the simulation of step t.

Thus, R' uses a constant number of steps to simulate a step of R and only $O(\log P(n)T(n))$ initialization time. Since $P(n) \leq 2^{T(n)}$, R' uses $O(T(n))$ steps to simulate $T(n)$ steps of R. □

Observation 1: R' needs only addition and subtraction to construct any address that it uses.

Observation 2: Each processor of R' uses only a constant number of local registers.

Chapter 4. Multiplication

In this chapter, we simulate a time-bounded PRAM[*] by four different models of computation: basic PRAM, unbounded fan-in circuit, bounded fan-in circuit, and Turing machine. We establish that polynomial time on a PRAM[*] or a PRAM and polynomial depth on a bounded or unbounded fan-in circuit and polynomial space on a TM are all equivalent.

4.1. Simulation of PRAM[*] by PRAM

Let R be a PRAM[*] operating in time $T(n)$ on inputs of length n and using at most $P(n)$ processors. Let R' be a PRAM[*] that uses only short addresses and simulates R according to the Associative Memory Lemma. Thus, R' uses $O(P^2(n)\,T(n))$ processors, $O(T(n))$ time, and only addresses in 0, 1, ..., $O(P(n)\,T(n))$. Each processor of R' uses only q registers, where q is a constant.

We construct a PRAM Z that simulates R via R' in $O(T^2(n) / \log T(n))$ time, using $O(P^2(n)\,T^2(n)\,n^2\,4^{T(n)} \log T(n))$ processors. We view Z as having $q + 4$ separate shared memories: mem_0, ..., mem_{q+3}. Our view facilitates description of the algorithm to follow. The idea of the proof is that Z stores the cell contents of R' with one bit per cell and acts as an unbounded fan-in circuit to manipulate the bits.

Initialization. Z partitions mem_q into $O(P(n)T(n))$ *sections* of $n2^{T(n)}$ cells each. Let $S(i)$ denote the ith section. A section is sufficiently long to hold any number generated in $T(n)$ steps by R', one bit per cell, in $n2^{T(n)}$-bit two's complement representation. Section $S(i)$ contains $con(i)$ of R' with one bit of $con(i)$ in each of the first $len(con(i))$ cells of the section. R' writes the more significant bits in cells with larger addresses.

Z partitions each of $mem_0, ..., mem_{q-1}$ into $O(P^2(n)T(n))$ *blocks* of $n2^{T(n)} \cdot T(n) \log T(n)$ sections each. Let $B_i(m)$ denote the mth block of mem_i. A block is large enough to implement the multiplication algorithm of Schönhage and Strassen (1971). The first section of $B_i(m)$ contains $rcon_m(i)$ of R' with one bit of $rcon_m(i)$ in each of the first $len(rcon_m(i))$ cells of the section.

Z activates $O(P^2(n)\,T(n))$ *primary processors*, one for each processor of R', in time $O(\log P(n)T(n))$. Z must quickly access individual cells in each block, so each primary processor activates $O(n^2\, 4^{T(n)}\, T(n) \log T(n))$ *secondary processors* in $O(T(n))$ time (Activation Lemma). For primary processor P_m, secondary processor P_j, $j \in \{m, ..., m+n2^{T(n)}-1\}$, assigns itself to the $(j-m)$th cell of the first section of a block. These processors will handle comparisons.

Secondary processors belonging to P_0 now construct a set of values to be used in the *SQUASH* procedure, to be defined later, which handles indirect addressing. The secondary processors for the first block set $con_{q+1}(i) = 2^i$, for all i, $0 \le i \le \log P(n)T(n)$.

In mem_{q+2}, Z builds an address lookup table containing the address of the first cell of $B_i(m)$ in cell m of the table, $0 \le m \le P(n)T(n)$. In all memories, the mth block begins at the same address. These addresses range up to $n^2 4^{T(n)} P(n)T^2(n) \log T(n)$, so Z creates the table in $O(T(n))$ time.

Z next spreads the input integer over the first n cells of $S(0)$ of mem_q, that is, Z places the jth bit of the input word in the jth cell of $S(0)$. This process takes constant time for processors $P_0, \cdots, P_{n-1}$, each performing the *BIT* instruction indexed by their processor number. (Note that without the $r(k) \leftarrow BIT(r(i))$ instruction, where $rcon_m(i) = j$, this process would take time $O(n)$ because for each j, $1 \le j \le n$, Z would have to construct a mask with a

1 in the jth position and 0's in all other positions to determine the value of the jth input bit. If $T(n) = o(n)$, then $O(n)$ time is unacceptably high.)

Simulation. We are now prepared to describe the simulation by Z of a general step of R'. Consider a processor P_g of R' and the corresponding primary processor P_m of Z. The actions of P_m and its secondary processors depend on the instruction executed by P_g of R'. P_m notifies its secondary processors of the instruction. The following cases arise.

$r(i) \leftarrow r(j) + r(k)$: Chandra *et al.* (1985) gave an unbounded fan-in circuit of size $O(x(\log^* x)^2)$ and constant depth for adding two integers of length x. Stockmeyer and Vishkin (1984) proved that an unbounded fan-in circuit of depth $D(n)$ and size $S(n)$ can be simulated by a CRCW PRAM in time $O(D(n))$ with $O(n+S(n))$ processors (Theorem 2.2). By the combination of these two results, the secondary processors perform addition in constant time with their concurrent write ability. This addition requires $O(n2^{T(n)}(\log^*(n2^{T(n)}))^2)$ processors.

$r(i) \leftarrow r(j) \wedge r(k)$: The secondary processors perform a Boolean AND in one step. Other Boolean operations are performed analogously.

$r(i) \leftarrow r(j) - r(k)$: The secondary processors add $rcon_g(j)$ and the two's complement of $rcon_g(k)$. This takes constant time.

Comparisons (*CJUMP* $r(i) > r(j)$, *label*): For $1 \le k \le n2^{T(n)}$, the secondary processor of the first section that normally handles the kth cell of the section handles the $(n2^{T(n)}-k+1)$th cell. Thus the lowest numbered processor reads the most significant bit. P_m writes a 0 in $c_{q+3}(m)$. All secondary processors read the $n2^{T(n)}$th cell in $B_i(g)$ and $B_j(g)$ to determine whether $rcon_g(i)$ and $rcon_g(j)$ are negative. If $rcon_g(i)$ ($rcon_g(j)$) is negative and the other is not, then P_m writes 2 (1) in $c_{q+3}(m)$. Otherwise, each secondary processor

allocated to the first section compares corresponding bits of $B_i(g)$ and $B_j(g)$. If both $rcon_g(i)$ and $rcon_g(j)$ are nonnegative, then if the bits are equal, it does nothing; if the bit of $B_i(g)$ is greater, it writes a 1 in $c_{q+3}(m)$; if the bit of $B_j(g)$ is greater, it writes a 2 in $c_{q+3}(m)$. If both $rcon_g(i)$ and $rcon_g(j)$ are negative, then if the bits are equal, it does nothing; if the bit of $B_i(g)$ is greater, it writes a 2 in $c_{q+3}(m)$; if the bit of $B_j(g)$ is greater, it writes a 1 in $c_{q+3}(m)$. After this step, if $rcon_g(i) = rcon_g(j)$, then $con_{q+3}(m) = 0$; if $rcon_g(i) \neq rcon_g(j)$, then $c_{q+3}(m)$ holds the value written by the lowest numbered secondary processor to write. If $con_{q+3}(m) = 1$, then the comparison is true; otherwise, the comparison is false. This process works by the concurrent write rules of the PRAM. Other comparisons are performed analogously and all comparisons can be simulated in constant time.

$r(i) \leftarrow r(j) * r(k)$: We use the following two lemmas.

Lemma 4.1.1. (Schönhage and Strassen, 1971) A log-space uniform, bounded fan-in circuit of depth $O(\log y)$ and size $O(y \log y \operatorname{loglog} y)$ can compute the product of two operands of length y.

Proof. For inputs of length y, Schönhage and Strassen (1971) gave a multiplication algorithm that may be implemented as a bounded fan-in circuit with depth $O(\log y)$ and size $O(y \log y \operatorname{loglog} y)$. □

Lemma 4.1.2. A log-space uniform, unbounded fan-in circuit of depth $O(\log y / \operatorname{loglog} y)$ and size $O(y^2 \log y \operatorname{loglog} y)$ can compute the product of two operands of length y.

Proof. Chandra *et al.* (1984) proved that for any $\varepsilon > 0$, a bounded fan-in circuit of depth $D(y)$ and size $S(y)$ can be simulated by an unbounded fan-in circuit of depth $O(D(y) / \varepsilon \operatorname{loglog} y)$ and size $O(2^{(\log y)^{\varepsilon}} \cdot S(y))$. Combining Lemma 4.1.1 with this result and setting $\varepsilon = 1$, we establish the lemma. □

R' can generate numbers of length up to $n2^{T(n)}$. By Theorem 2.2 and Lemma 4.1.2, a CRCW PRAM can simulate a bounded fan-in circuit performing multiplication in time $O(T(n) / \log T(n))$ with $O(n^2\, 4^{T(n)}\, T(n) \log T(n))$ processors.

Indirect addressing: By the Associative Memory Lemma, R' accesses only addresses of length $O(\log P(n)T(n))$. If P_g wishes to perform an indirect read from $c(r(i))$, then P_m and its associated processors perform a *SQUASH* on $B_i(g)$ in time $O(\text{loglog } P(n)T(n))$. The goal of *SQUASH* is to place in a single cell the integer whose two's complement representation is stored in a section with one bit per cell. In a *SQUASH*, the secondary processors associated with the contents of the first $O(\log P(n)T(n))$ cells of a block read their cells. If the kth cell contains a 0, then the kth processor sets $x_k := 0$. If the kth cell contains a 1, then the kth processor sets $x_k := 2^k$. (Recall that 2^k was previously computed during the initialization period.) Next, the secondary processors compute $\sum_{k=0}^{\log P(n)T(n)} x_k$ in $O(\text{loglog } P(n)T(n)) = O(\log T(n))$ time since $P(n) \le 2^{T(n)}$. *SQUASH* places a number, which was stored one bit per cell in the first $O(\log P(n)T(n))$ cells of a block, into a single cell.

If processors P_f and P_g of R' wish to simultaneously write $c(j)$, then the corresponding processors P_l and P_m of Z will simultaneously attempt to write $S(j)$ of mem_q. If $f < g$, then $l < m$, and all secondary processors of P_l are numbered less than all secondary processors of P_m. Thus, in R', P_f succeeds in its write, and in Z, P_l and its secondary processors succeed in their writes.

Theorem 4.1. For all $T(n) \ge \log n$, $PRAM[*]\text{–}TIME(T(n)) \subseteq$ $PRAM\text{–}TIME(T^2(n) / \log T(n))$.

Proof. According to the above discussion, Z simulates R via R'. Initialization takes $O(\log(P(n)T(n)) + T(n) + \log n) = O(T(n))$ time. Z performs indirect addressing in $O(\log\ T(n))$ time, multiplication in $O(T(n) / \log T(n))$ time, and all other operations in constant time. Thus, Z uses time $O(T(n) / \log T(n))$ to simulate each step of R'. Z uses $O(P^2(n)T(n))$ primary processors, each with $O(n^2\ 4^{T(n)}\ T(n) \log T(n))$ secondary processors. Hence, Z simulates R in $O(T^2(n) / \log T(n))$ time, using $O(P^2(n)\ T^2(n)\ n^2\ 4^{T(n)} \log T(n))$ processors. □

Corollary 4.1.1. *PRAM* [*]*–PTIME = PRAM –PTIME.*

Corollary 4.1.2. *PRAM* [*]*–POLYLOGTIME = PRAM –POLYLOGTIME.*

If $T(n) = O(\log n)$, then $P(n)$ is a polynomial in n, and Z simulates R in time $O(\log^2 n$ / loglog $n)$ with polynomially many processors. Thus, an algorithm running in time $O(\log n)$ on a PRAM[*] is in NC^2. If $T(n) = O(\log^k n)$, then Z simulates R in time $O(\log^{2k} n$ / $(2k$ loglog $n))$ with $O(n^{2+4\log^{k-1} n} \cdot \log^{2k} n$ loglog $n)$ processors. So, our simulation does *not* show that an algorithm running in time $O(\log^k n)$, $k>1$, on a PRAM[*] is in NC because of the superpolynomial processor count. An interesting open problem is to show either that *PRAM* [*]*–POLYLOGTIME = NC* by reducing the processor count to a polynomial or that NC is strictly included in *PRAM* [*]*–POLYLOGTIME* by proving that the simulation requires a superpolynomial number of processors.

4.2. Simulations of PRAM[*] by Circuits and Turing Machine

We now describe simulations of a PRAM[*] by a log-space uniform family of unbounded fan-in circuits, a log-space uniform family of bounded fan-in circuits, and a Turing machine.

Lemma 4.2.1. For each n, every language recognized by a PRAM[*] R in time $T(n)$ with $P(n)$ processors can be recognized by a log-space uniform, unbounded fan-in circuit UC_n of depth $O(T^2(n) / \log T(n))$ and size $O(n^2 T^2(n) 8^{T(n)} \log T(n))$.

Proof. The depth bound follows from Theorems 4.1 and 2.1. We now establish the size bound. Let R' be the PRAM[*] described in Theorem 4.1 that simulates R according to the Associative Memory Lemma, using $O(T(n))$ time with $O(P^2(n) T(n))$ processors. Fix an input length n. Let UC_n be a log-space uniform, unbounded fan-in circuit that simulates R' by the construction given by Stockmeyer and Vishkin (1984) (Theorem 2.1), with one modification. For each time step of R', we add to UC_n a block of depth $O(T(n) / \log T(n))$ and size $O(n^2 4^{T(n)} T(n) \log T(n))$ that handles multiplication (Lemma 4.1.2). Thus, UC_n has depth $O(T^2(n) / \log T(n))$ and size $O(P(n)T(n)[T(n)(n+T(n)) + (n+T(n))^3 + (n+T(n))(n+P(n)T(n)) + n^2 4^{T(n)} T(n) \log T(n)]) = O(n^2 T^2(n) 8^{T(n)} \log T(n))$, since $P(n) \leq 2^{T(n)}$. □

Lemma 4.2.2. For each n, every language recognized by a PRAM[*] R in time $T(n)$ with $P(n)$ processors can be recognized by a log-space uniform, bounded fan-in circuit BC_n of depth $O(T^2(n))$ and size $O(n^2 T^2(n) 8^{T(n)} \log T(n))$.

Proof. Fix an input length n. Let UC_n be the unbounded fan-in circuit described in Lemma 4.2.1 that simulates R. Except for the circuit blocks implementing multiplication, the portions of the circuit that simulate a single time step of S' have constant depth and fan-in at most $O(P^2(n) T(n))$. Hence, these parts of the circuit can be implemented as a bounded fan-in circuit of depth $O(\log P(n) + \log T(n))$. The multiplication blocks may be implemented as bounded fan-in circuits of depth $O(T(n))$ (Lemma 4.1.1). Let BC_n be this bounded fan-in implementation of UC_n. Since $P(n) \leq 2^{T(n)}$, BC_n simulates each step of S'

in depth $O(T(n))$, and hence BC_n simulates S via S' in depth $O(T^2(n))$ and size $O(n^2\, T^2(n)\, 8^{T(n)} \log T(n))$. □

Theorem 4.2. For all $T(n) \geq \log n$, $PRAM[*]\text{-}TIME(T(n)) \subseteq DSPACE(T^2(n))$.

Proof. Theorem 4.2 follows from Lemma 4.2.2 and Borodin's (1977) result that a log-space uniform, bounded fan-in circuit of depth $D(n)$ can be simulated in space $O(D(n))$ on a Turing machine. □

Corollary 4.2.1. $PRAM[*]\text{-}PTIME = PSPACE$.

4.3. Direct Simulation of PRAM[*] by Turing Machine

For the sake of completeness, we describe a simulation of PRAM[*] R via R' by a deterministic Turing machine M that uses space $T^2(n)$. This is a direct simulation by the TM, rather than through circuits, as in Section 4.2. M simulates R' by writing only one bit at a time of a cell's contents. Let λ denote the empty string.

Let R be a PRAM[*] operating in time $T(n)$ on inputs of length n and using at most $P(n)$ processors. Let R' be a PRAM[*] that accesses only short addresses and simulates R according to the Associative Memory Lemma. Thus, R' uses $O(P^2(n)\, T(n))$ processors, $O(T(n))$ time, and only addresses in 0, 1, ..., $O(P(n)\, T(n))$. Each processor of R' uses only q registers, where q is a constant.

We construct a deterministic Turing machine M that simulates R via R' in space $T^2(n)$. By the definition of PRAM, R' accepts input ω if and only if, by the execution of R' on ω, P_0 executes an *ACCEPT* instruction. To check this condition, M executes the mutually recursive procedures $PCOUNTER(m, t)$, which returns the contents of the program counter

of P_m at time t, and *VALUE* (b, i, m, t), which returns the bth bit of $rcon_m(i)$ at time t, if $m \neq \lambda$, or the bth bit of $con(i)$ at time t, if $m = \lambda$. (*VALUE* is based on the procedure *FIND* in Hartmanis and Simon (1974).)

PCOUNTER (m, t) works as follows. Let p be the value returned by *PCOUNTER* (m, t). *PCOUNTER* (m, t) depends on r, the value returned by *PCOUNTER* $(m, t-1)$. If r indicates that P_m was not active at time $t-1$, then *PCOUNTER* determines by calls to *PCOUNTER* for time $t-1$ whether $P_{\lfloor m/2 \rfloor}$ activated P_m at time $t-1$ with a *FORK* instruction. If $P_{\lfloor m/2 \rfloor}$ executes *FORK label* 1, *label* 2, then if m is even, $p =$ *label* 1; otherwise, $p =$ *label* 2. If r indicates that P_m is active at time $t-1$ and step r is not a *CJUMP*, *REJECT*, or *ACCEPT* instruction, then $p = r+1$. If step r is *CJUMP* $r(i)$ *comp* $r(j)$, *label* 3, where *comp* is an integer comparison, then *PCOUNTER* repeatedly calls *VALUE* for time $t-1$ to compare $rcon_m(i)$ and $rcon_m(j)$. If the comparison is true, then $p =$ *label* 3; otherwise, $p = r+1$. If instruction r is an *ACCEPT* (*REJECT*), then $p = r$.

VALUE (b, i, m, t) works as follows. Let v denote the output of *VALUE* (b, i, m, t). Suppose $m = \lambda$. By calls to *PCOUNTER*, M determines whether some processor wrote $c(i)$ at time $t-1$. If no processor wrote $c(i)$, then $v =$ *VALUE* $(b, i, \lambda, t-1)$. Otherwise, suppose P_m executed instruction $c(r(k)) \leftarrow r(j)$ such that $rcon_m(k) = i$ and was the lowest numbered processor that wrote $c(i)$ at time $t-1$. Then $v =$ *VALUE* $(b, j, m, t-1)$.

Suppose $m \neq \lambda$. By calls to *PCOUNTER*, M determines whether P_m wrote $r_m(i)$ at time $t-1$. If not, then $v =$ *VALUE* $(b, i, m, t-1)$. Otherwise, suppose P_m executed instruction *instr* that wrote $r_m(i)$ at time $t-1$.

If *instr* is $r(i) \leftarrow k$ for a constant k, then v is bit b of k.

If *instr* is $r(i) \leftarrow r(j)$, then $v = VALUE(b, j, m, t-1)$.

If *instr* is a Boolean operation, such as $r(i) \leftarrow r(j) \wedge r(k)$, then v is the result of the Boolean operation on the bth bits of the operands, in this case, $v = VALUE(b, j, m, t-1) \wedge VALUE(b, k, m, t-1)$. M handles the other Boolean operations similarly.

If *instr* is an arithmetic instruction rather than a Boolean instruction, then the execution of *VALUE* is more complicated. Suppose *instr* is a multiplication instruction such as $r(i) \leftarrow r(j) * r(k)$. Let $w := \max\{len(rcon_m(j)), len(rcon_m(k))\}$ be the length of the longer operand. Then $rcon_m(i)$ is the sum of at most w partial products, each of length at most $2w$. The value of bit b of $rcon_m(i)$ depends on bit b of each partial product and on the carry of length at most $\log w$ from column $b-1$. Since w can be as large as $2^{T(n)}$, M cannot write all w partial products in polynomial space. So, M computes the sum of the w partial products that contribute to bit b as follows. M computes the carry from column $b-1$ by computing the sum of each column and the carry from each column from right to left starting at the rightmost column. Each partial product is the product of $rcon_m(j)$ and a bit of $rcon_m(k)$. M uses an accumulator to keep a running total of the sum of the partial products in each column. Suppose M is computing the sum of the sth column. M uses two pointers: one points to the bit of $rcon_m(j)$ involved in the sth column of the gth partial product, the other to the bit of $rcon_m(k)$ involved in the sth column of the gth partial product. Recursive calls to *VALUE* return the values of these bits. M computes their product (Boolean AND) and adds the product to the accumulator. M then shifts the pointers to point to the bits involved in the sth column of the $(g+1)$st partial product, finds these bits, computes their product and adds it

to the accumulator, and so on until the sth column of all w terms have been summed. The carry from column s to column $s+1$ equals the sum of the sth column shifted right by one bit position. This process continues until the sum of column b is computed; the value of the bth bit of $rcon_m(i)$ is the lowest-order (rightmost) bit of this sum.

If the multiplier, $rcon_m(k)$, is negative, then M multiplies $rcon_m(j)$ by $|rcon_m(k)|$ and adjusts the sign at the end.

If *instr* is an addition or a subtraction instruction, then M's actions are similar to, but simpler than, its actions for a multiplication instruction.

If *instr* is an indirect write, such as $r(i) \leftarrow r(r(j))$, then, by calls to *VALUE* for time $t-1$, M obtains the bits of $rcon_m(j)$. Then $v = VALUE(b, rcon_m(j), m, t-1)$. If *instr* is $r(i) \leftarrow c(r(j))$, then $v = VALUE(b, rcon_m(j), \lambda, t-1)$. By the Associative Memory Lemma, M can write $rcon_m(j)$ in $O(T(n))$ space.

Theorem 4.3. For all $T(n) \geq \log n$, $PRAM[*]\text{-}TIME(T(n)) \subseteq DSPACE(T^2(n))$.

Proof. M simulates R via R' by the simulation described above.

The input is n bits long and the length of the contents of any cell may at most double at each step, so the length of each cell's contents can grow to at most $n2^{T(n)}$ bits. Hence, M can write b, the bit number, in $O(T(n))$ space. M can write i, the cell address, in $O(\log P(n)T(n))$ space. Since $P(n) \leq 2^{T(n)}$, $O(\log P(n)T(n)) = O(T(n))$. R' activates at most $O(P^2(n)T(n))$ processors, so M can write m, the processor number, in $O(log\ (P^2(n)T(n))) = O(T(n))$ space. M can write t, the time step, in $O(\log T(n))$ space. Therefore, M can write the parameters of *VALUE* and *PCOUNTER* in $O(T(n))$ space.

Consider the space M requires to execute the simulation. M writes all the variables used in one invocation of *VALUE* or *PCOUNTER* in space $O(T(n))$, the same space required

to write the parameters of *VALUE* or *PCOUNTER*. If an instance of *VALUE* or *PCOUNTER* with time parameter t makes a recursive call to *VALUE* or *PCOUNTER*, then the called instance will have time parameter $t-1$. Recall that R' simulates R in time $O(T(n))$, so the depth of recursion is $O(T(n))$. Hence, with linear space compression, M simulates R via R' in space $T^2(n)$. □

Chapter 5. Division

In this chapter, we study the division instruction. Let us assume that the division instruction returns the quotient. We are interested in the division instruction for two reasons. First, division is a natural arithmetic operation. Second, Simon (1981a) has shown that a RAM with division and left shift ($\uparrow$) can be very powerful. He proved $RAM[\uparrow,+]\text{-}PTIME = ER$, where ER is the class of languages accepted in time

$$2^{2^{2^{\cdots^{2}}}} \Big\uparrow n$$

by Turing machines. At first glance, this is a surprising result: for a RAM with left shift and right shift ($\downarrow$), we already know that $RAM[\uparrow,\downarrow]\text{-}PTIME = PSPACE$ (Simon 1977). Simon proved $ER \subseteq RAM\text{-}[\uparrow,+]\text{-}PTIME$ by building very long integers with the left shift operation and then manipulating them as both integers and binary strings. The division instruction is used to generate a complex set of strings representing all possible TM configurations. (Note that right shift cannot replace division in building these strings.)

We consider the power of division paired with multiplication rather than with left shift, as well as the power of only division with our basic instruction set. From Hartmanis and Simon (1974), we also know that $RAM[*,+]\text{-}PTIME = PSPACE$.

In the following, let MD be a PRAM[*,÷] that uses $T(n)$ time and $P(n)$ processors. Let MD' be a PRAM[*,÷] that uses only short addresses and simulates MD according to the Associative Memory Lemma. Thus, MD' uses $O(P^2(n)T(n))$ processors, $O(T(n))$ time, and only addresses in 0, 1, ..., $O(P(n)T(n))$.

We begin by describing the simulation of a PRAM[*,÷] by a PRAM. The idea of the proof is that we modify the simulation of a PRAM[*] by a PRAM (Section 4.1). Because

this simulation depends on the relationship between circuits and PRAMs (Theorem 2.2), we are interested in the Boolean circuit complexity of division. Beame *et al.* (1986) developed a circuit for dividing two n-bit numbers in depth $O(\log n)$. This circuit, however, is polynomial-time uniform, and we need the stronger condition of log-space uniformity. Reif (1986) devised a log-space uniform, depth $O(\log n \log\log n)$ division circuit, and Shankar and Ramachandran (1987) improved the size bound of this circuit. We will need the following lemma.

Lemma 5.1.1. A PRAM can compute the quotient of two x bit operands in time $O(\log x)$ with $O((1/\delta^4)\, x^{1+\delta})$ processors, for any $\delta > 0$.

Proof. Given in Shankar and Ramachandran (1987). □

Simulation. We construct a PRAM Z that simulates MD via MD' in time $O(T^2(n))$. We modify the simulation of a PRAM[*] by a PRAM (Section 4.1) to deal with division instructions. By Lemma 5.1.1 with $x = n\,2^{T(n)}$, Z can perform a division in time $O(T(n))$ with the available processors and $0 < \delta \leq 1$.

Theorem 5.1. For all $T(n) \geq \log n$, $PRAM[*,+]\text{-}TIME(T(n)) \subseteq PRAM\text{-}TIME(T^2(n))$.

Proof. By the simulation above, Z simulates each step of MD' in time $O(T(n))$ with $O(P^2(n)\, T^2(n)\, n^2\, 4^{T(n)} \log T(n))$ processors. MD' runs for $O(T(n))$ steps, so Z can simulate MD via MD' in $O(T^2(n))$ steps. □

Corollary 5.1.1. $PRAM[*,+]\text{-}PTIME = PRAM\text{-}PTIME$.

Corollary 5.1.2. $PRAM[*,+]\text{-}POLYLOGTIME = PRAM\text{-}POLYLOGTIME$.

Next, we consider the simulation of a PRAM[*,+] by a Turing machine. We construct a TM M that simulates MD via MD' in $T^2(n) \log T(n)$ space by modifying the simulation of a PRAM[*] by a TM (Section 4.2).

We will need the following lemmas.

Lemma 5.2.1. A log-space uniform, bounded fan-in circuit can compute the quotient of two x bit operands in depth $O(\log x \log\log x)$ and size $O((1/\delta^4)\, x^{1+\delta})$, for any $\delta > 0$.

Proof. Given in Shankar and Ramachandran (1987). □

Lemma 5.2.2. For each n, every language recognized by a PRAM[*,÷] MD in time $T(n)$ with $P(n)$ processors can be recognized by a log-space uniform bounded fan-in circuit DC_n of depth $O(T^2(n) \log T(n))$.

Proof. Fix an input length n. Let BC_n be the bounded fan-in circuit described in Lemma 4.2.2 that simulates a PRAM[*]. Let DC_n be BC_n with additional circuit blocks for division. To handle division instructions with operands of length at most $x = n\,2^{T(n)}$, we use the log-space uniform $O(\log x \log\log x)$ depth bounded fan-in division circuit specified in Lemma 5.2.1. Circuit DC_n is at most at constant factor larger in size than BC_n. Hence, DC_n uses depth $O(T(n) \log T(n))$ to simulate each step of MD. □

Theorem 5.2. For all $T(n) \geq \log n$, $PRAM\,[*,\div]\text{–}TIME\,(T(n)) \subseteq DSPACE\,(T^2(n) \log T(n))$.

Proof. Theorem 5.2 follows from Lemma 5.2.2 and Borodin's (1977) result that a bounded fan-in circuit of depth $D(n)$ can be simulated in space $O(D(n))$ on a Turing machine. □

Through Theorem 5.2 and the simulation of $DSPACE\,(T(n))$ in $PRAM\text{–}TIME\,(T(n))$ (Fortune and Wyllie, 1978), we can obtain an $O(T^2(n) \log T(n))$ time simulation of a PRAM[*,÷] by a PRAM. We remove the $\log T(n)$ factor by the direct simulation in Theorem 5.1.

Through Theorem 5.1 and the simulation of $PRAM\text{–}TIME\,(T(n))$ in $DSPACE\,(T^2(n))$ (Fortune and Wyllie, 1978), we obtain an $O(T^4(n))$ space simulation of a PRAM[*,÷] by a TM. The simulation of Theorem 5.2 is more efficient.

Corollary 5.2.1. *PRAM* [*, +]*–PTIME = PSPACE.*

We now present the simulation of a PRAM[÷] by a PRAM. Let D be a PRAM[÷] that uses $T(n)$ time and $P(n)$ processors. We construct a PRAM Z that simulates D in time $O(T(n) \log (n+T(n)))$. Z acts as a circuit to simulate the computation of D.

Simulation. We modify the simulation of a PRAM[*] by a PRAM from Section 4.1. In $T(n)$ steps, a PRAM[*] can build integers of length $n2^{T(n)}$, whereas a PRAM[÷] can build only integers of length $O(n+T(n))$. As a result, Z partitions the memory into blocks containing only $O(n+T(n))$ cells each. Z activates $P(n)$ primary processors, each with $O((1/\delta^4)(n+T(n))^{1+\delta})$ secondary processors. The simulation proceeds along the same lines as in Section 4.1 until a division instruction arises. By Lemma 5.1.1, Z can perform a division in time $O(\log (n+T(n)))$.

Theorem 5.3. For all $T(n) \geq \log n$, *PRAM* [÷]*–TIME* $(T(n)) \subseteq$ *PRAM –TIME* $(T(n) \log (n+T(n)))$.

Proof. By the simulation above, Z simulates D in time $O(T(n) \log (n+T(n)))$ with $O((P(n)/\delta^4)(n+T(n))^{1+\delta})$ processors. □

Corollary 5.3.1. *PRAM* [÷]*–PTIME = PRAM –PTIME.*

Corollary 5.3.2. *PRAM* [÷]*–POLYLOGTIME = PRAM –POLYLOGTIME.*

Observe that a RAM[÷] is unable to quickly generate long integers. Therefore, the gap between the time-bounded power of a RAM[÷] and the time-bounded power of a PRAM[÷] is much greater than the gap between the power of a RAM[*] and the power of a PRAM[*].

Let $PC = \{PC_1, PC_2, ...\}$ be the family of bounded fan-in circuits that simulates the family C of unbounded fan-in circuits described by Stockmeyer and Vishkin (1984). For a fixed input size n, the depth of PC_n is $O(T(n) \log P(n)T(n))$ and the size is $O(P(n)T(n)[T(n)(n+T(n)) + (n+T(n))^3 + (n+T(n))(n+P(n)T(n))])$.

Theorem 5.4. For each n, every language recognized by a PRAM[+] D in time $T(n)$ with $P(n)$ processors can be recognized by a log-space uniform, bounded fan-in circuit DB_n of depth $O(T(n) \log P(n) + T(n) \log(n+T(n)) \text{loglog}(n+T(n)))$.

Proof. Fix an input length n. Let PC_n be the bounded fan-in circuit described above that simulates a PRAM. Let DB_n be PC_n with additional circuit blocks for division. To handle division instructions with operands of length at most $x = n + T(n)$, we use the log-space uniform, $O(\log x \text{ loglog } x)$ depth, bounded fan-in division circuit specified in Lemma 5.2.1. Circuit DB_n is at most at constant factor larger in size than PC_n. Hence, DB_n uses depth $O(\log P(n) + \log(n+T(n)) \text{ loglog } (n+T(n)))$ to simulate each step of D. □

Lemma 5.5.1. An off-line Turing machine can compute the quotient of two n bit operands in $O(\log n \text{ loglog } n)$ space.

Proof. Borodin (1977) proved that an off-line TM can simulate a log-space uniform circuit with bounded fan-in and depth $D(n)$ in space $O(D(n))$. Combined with the log-space uniform $O(\log n \text{ loglog } n)$ depth division circuit of Shankar and Ramachandran (1987), we have the lemma. □

Theorem 5.5. For all $T(n) \geq \log n$, $PRAM[+]\text{-}TIME(T(n)) \subseteq DSPACE(T^2(n))$.

Proof. Fortune and Wyllie (1978) simulated each PRAM running in time $T(n)$ by a TM running in space $O(T^2(n))$. They used recursive procedures of depth $O(T(n))$ using space $O(T(n))$ at each level of recursion. If we augment the simulated PRAM with division, then

by Lemma 5.5.1, an additional $O(\log(n+T(n))\ \text{loglog}(n+T(n)))$ space is needed at each level, so $O(T(n))$ space at each level still suffices. Hence, with linear space compression, a TM with space $T^2(n)$ can simulate a PRAM[+] running in time $T(n)$. □

Corollary 5.5.1. *PRAM* [+]*–PTIME = PSPACE.*

Chapter 6. Shift

Pratt and Stockmeyer (1976) proved that for vector machines, that is, a $RAM[\uparrow,\downarrow]$ without addition or subtraction in which left shift ($\uparrow$) and right shift ($\downarrow$) distances are restricted to a polynomial number of bit positions, $RAM[\uparrow,\downarrow]$-$PTIME = PSPACE$. Simon (1977) proved the same equality for RAMs with *unrestricted* left shift and right shift, addition, and subtraction. We prove that polynomial time on PRAMs with unrestricted shifts is equivalent to polynomial time on basic PRAMs and to polynomial space on Turing machines (TMs).

6.1. Simulation of PRAM[$\uparrow$,$\downarrow$] by PRAM

By repeated application of the left shift instruction, a PRAM[$\uparrow$,$\downarrow$] can generate numbers of length

$$O\left[2^{2^{\cdots^{2}}}\right] \quad T\updownarrow(n)$$

in $T(n)$ steps. These extremely large numbers will contain very long strings of 0's, however. (If Boolean operations are used, then the numbers will have very long strings of 0's and very long strings of 1's.) Since we cannot write such numbers in polynomial space, nor can we address an individual bit of such a number in polynomial space, we encode the numbers and manipulate the encodings. We use the *marked interesting bit (MIB) encoding*, an enhancement of the interesting bit encoding of Simon (1977). Let d be an integer. Define $len(d)$ as the minimum integer w such that $-2^{w-1} \le d \le 2^{w-1}-1$. Let $b_{w-1} \cdots b_0$ be the w-bit two's complement representation of d. An *interesting bit* of d is a bit b_i such that $b_i \neq b_{i+1}$. (b_w is not an interesting bit.)

If d has an interesting bit at b_i and the next interesting bit is at b_j, $i < j$, then the bits $b_j b_{j-1} \cdots b_{i+1}$ are identical. If these bits are 0's (1's), then we say that d has a *constant interval* of 0's (1's) at b_j.

If a constant interval has length 1, then the entire interval is a single bit, which is an interesting bit. We call such an interesting bit a *singleton*. We mark interesting bits that are singletons. We define the MIB encoding as

$$E(0) = 0s,$$

$$E(01) = 1s,$$

$$E(d) = (E(a_t)q_t, \ldots, E(a_2)q_2, E(a_1)q_1; r),$$

where d is an integer; a_j is the position of the jth interesting bit of d; $q_j = s$ if the jth interesting bit is a singleton and $q_j = m$ if the jth interesting bit is not a singleton; and r is the value (0 or 1) of the rightmost bit of d. Call q_j the *mark* of the jth interesting bit. Call r the *start bit*.

For example, $E(01100) = (E(011)m, E(01)m; 0) = ((E(01)m; 1)m, 1sm; 0) = ((1sm; 1)m, 1sm; 0)$. For all d, define $val(E(d)) = d$. The marks of interesting bits will be useful for quickly computing $E(d+1)$ from $E(d)$.

An encoding can be viewed as a tree. For the encoding $E(d) = (E(a_t)q_t, \ldots, E(a_2)q_2, E(a_1)q_1; r)$, a *node* is associated with each of $E(a_t)q_t, \ldots, E(a_1)q_1, r$. A root node is associated with the entire encoding of $E(d)$ and holds nothing. A nonroot node holds one of: $0s$; $1s$; r, the start bit; or q_j, the mark of the jth interesting bit. If a node holds $0s$, $1s$, or r, then it is a leaf. If a node holds q_j, then it is an internal node, and its children are the nodes of $E(a_j)$. Figure 6.1 contains a sketch of the encoding tree of E(01100).

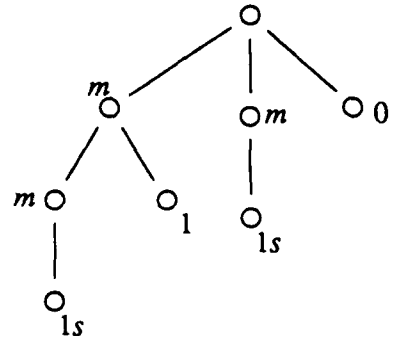


Figure 6.1. Encoding tree for E(01100)

For a node α corresponding to $E(a_k)q_k$, the value of the subtree rooted at α, $val(\alpha)$, is a_k, the value of $E(a_k)$. Thus, $val(\alpha)$ is the position of an interesting bit.

We define *level 0* of a tree as the root. We define *level j* of a tree as the set of all children of nodes in level $j-1$ of the tree.

A *pointer* into an encoding specifies a path starting at the root of the tree. For instance, the pointer 7.5.9 specifies a path x_0,x_1,x_2,x_3 in which x_0 is the root, x_1 is the 7th child (from the right) of x_0, x_2 is the 5th child of x_1, and x_3 is the 9th child of x_2. A pointer also specifies the subtree rooted at the last node of the path.

For an integer d, suppose $E(d) = (E(a_t)q_t, ..., E(a_1)q_1;r)$. We define $intbits(d) = t$, the number of interesting bits in d. Viewing $E(d)$ as a tree, we refer to $E(a_s)$ as a *subtree* of $E(d)$. We define the *k th subtree at level c* of $E(d)$ as the k th subtree from the right whose root is distance c from the root of $E(d)$. We define $depth(d)$ recursively by

$$depth(0) = depth(1) = 1,$$

$$depth(d) = 1 + \max\{depth(a_t), ..., depth(a_1)\}.$$

We now present three lemmas, analogous to those of Simon (1977), that bound the length of a pointer into an encoding. Lemma 6.1.1 bounds the depth of an encoding and the number of interesting bits in a number generated by a PRAM[↑,↓]. Let *bool* be a set of Boolean operations.

Lemma 6.1.1. Suppose a processor P_m executes $r(i) \leftarrow r(j)\ O\ r(k)$, $O \in \{+, \uparrow, \downarrow, -,$ *bool* $\}$.

i) If O is +, then $depth(rcon_m(i)) \leq 1 + \max\{depth(rcon_m(j)), depth(rcon_m(k))\}$ and $intbits(rcon_m(i)) \leq intbits(rcon_m(j)) + intbits(rcon_m(k))$.

ii) If O is a Boolean operation, then $depth(rcon_m(i)) \leq \max\{depth(rcon_m(j)), depth(rcon_m(k))\}$ and $intbits(rcon_m(i)) \leq intbits(rcon_m(j)) + intbits(rcon_m(k))$.

iii) If O is −, then $depth(rcon_m(i)) \leq 1 + \max\{depth(rcon_m(j)), depth(rcon_m(k))\}$ and $intbits(rcon_m(i)) \leq intbits(rcon_m(j)) + intbits(rcon_m(k))$.

iv) If O is ↑ or ↓, then $depth(rcon_m(i)) \leq 2 + \max\{depth(rcon_m(j)), depth(rcon_m(k))\}$ and $intbits(rcon_m(i)) \leq 1 + intbits(rcon_m(j))$.

Proof. i) If P_m executes $r(i) \leftarrow r(j) + r(k)$, then an interesting bit of $rcon_m(i)$ is either in the same position or is one position to the left of an interesting bit of $rcon_m(j)$ or $rcon_m(k)$. As a result, depth may increase by at most 1. For example, using two's complement representations, if $rcon_m(j) = 011$ and $rcon_m(k) = 01$, then $rcon_m(i) = 0100$. The depth of both addends is 1, while the depth of their sum is 2. Thus, $depth(rcon_m(i))$ will be at most $1 + \max\{depth(rcon_m(j)), depth(rcon_m(k))\}$, and $intbits(rcon_m(i))$ will be at most $intbits(rcon_m(j)) + intbits(rcon_m(k))$.

ii) If P_m executes $r(i) \leftarrow r(j) \vee r(k)$, then an interesting bit of $rcon_m(i)$ is an interesting bit of either $rcon_m(j)$ or $rcon_m(k)$. Thus, $depth(rcon_m(i))$ will be at most $\max\{depth(rcon_m(j)), depth(rcon_m(k))\}$, and $intbits(rcon_m(i))$ will be at most $intbits(rcon_m(j)) + intbits(rcon_m(k))$. Other Boolean operations produce the same results.

iii) If P_m executes $r(i) \leftarrow r(j) - r(k)$, and we view $r(j) - r(k)$ as $r(j) + (1 + \neg r(k))$, then by Parts i) and ii), $depth(rcon_m(i))$ will be at most $1 + \max\{depth(rcon_m(j)), depth(rcon_m(k))\}$, and $intbits(rcon_m(i))$ will be at most $intbits(rcon_m(j)) + intbits(rcon_m(k))$.

iv) If P_m executes $r(i) \leftarrow r(j) \uparrow r(k)$ and $E(rcon_m(j)) = (E(a_q), ..., E(a_1); w)$, $w \in \{0,1\}$, then $E(rcon_m(i)) = (E(a_q + rcon_m(k)), ..., E(a_1 + rcon_m(k)), [E(rcon_m(k))]; x)$, where we include $E(rcon_m(k))$ if and only if $rcon_m(k) = 0$ and $w = 1$ $x = w$ if $rcon_m(k) = 0$, $x = 0$ if $rcon_m(k) > 0$. The claim holds. By a similar argument, the claim holds for right shift. □

Part i) of Lemma 6.1.2 bounds the number of subtrees below first level nodes in an encoding; Part ii) bounds the number of subtrees below f th level nodes in an encoding, $f > 1$.

Lemma 6.1.2. Suppose a processor P_m executes $r(i) \leftarrow r(j)$ O $r(k)$, where O $\in \{+, \uparrow, \downarrow, -, bool\}$, $E(rcon_m(i)) = (E(a_r), ..., E(a_1); w_i)$, $E(rcon_m(j)) = (E(b_s), ..., E(b_1); w_j)$, and $E(rcon_m(k)) = (E(c_t), ..., E(c_1); w_k)$, where a_v, b_v, and c_v denote the positions of the v th interesting bits of $rcon_m(i)$, $rcon_m(j)$, and $rcon_m(k)$, respectively.

i) For $E(a_v)$ (that is, the v th subtree at level 1 of $E(rcon_m(i))$),

a) if O is +, then $intbits(a_v) \leq 1 + \max_q \{intbits(b_q), intbits(c_q)\}$,

b) if O is $\uparrow$ or $\downarrow$, then $intbits(a_v) \leq \max_q \{intbits(b_q)\} + intbits(rcon_m(k))$, and

c) if O is a Boolean operation, then $intbits(a_v) \leq \max_q \{intbits(b_q), intbits(c_q)\}$,

and

d) if $\bigcirc$ is $-$, then $intbits(a_v) \leq 1 + \max_q \{intbits(b_q), intbits(c_q)\}$.

ii) For $E(\beta)$ a subtree at level $f > 1$, $intbits(\beta) \leq 1 + \max_q \{intbits(q\text{ th subtree of } rcon_m(j) \text{ at level } f), intbits(q\text{ th subtree of } rcon_m(k) \text{ at level } f)\}$.

Proof. i) A first level subtree $E(a_v)$ encodes the position p of an interesting bit in $rcon_m(i)$. The subtrees of $E(a_v)$ encode the positions of interesting bits in p.

a) If P_m executes $r(i) \leftarrow r(j) + r(k)$, then, as can be seen in ADD, p is either the position of an interesting bit in $rcon_m(j)$ or $rcon_m(k)$ or p is one plus the position of an interesting bit in $rcon_m(j)$ or $rcon_m(k)$. In the first case, $intbits(a_v) \leq \max_q \{intbits(b_q), intbits(c_q)\}$. In the second case, by Lemma 6.1.1 i), $intbits(a_v) \leq 1 + \max_q \{intbits(b_q), intbits(c_q)\}$.

b) If P_m executes $r(i) \leftarrow r(j) \uparrow r(k)$, then p is the sum of $rcon_m(k)$ and the position of an interesting bit of $rcon_m(j)$. By Lemma 6.1.1, $intbits(a_v) \leq \max_q \{intbits(b_q)\} + intbits(rcon_m(k))$. By a similar argument, the claim holds for right shift.

c) If P_m executes $r(i) \leftarrow r(j) \vee r(k)$, then p is the position of an interesting bit in either $rcon_m(j)$ or $rcon_m(k)$. Thus, $intbits(a_v) \leq \max_q \{intbits(b_q), intbits(c_q)\}$. Other Boolean operations produce the same results.

d) If P_m executes $r(i) \leftarrow r(j) - r(k)$, then the claim holds by Parts i)a) and i)c).

ii) For any instruction, we add at most 1 to the value of a subtree of level $f > 1$; hence, Part i)a) applies. □

Lemma 6.1.3. A pointer can be specified in $O(T^2(n))$ space on a TM.

Proof. Let d be an integer generated by a PRAM[↑,↓]. By Lemma 6.1.1, $depth(d) \leq 2T(n)$. If ω is the input to the PRAM[↑,↓] and ω has length n, then $intbits(\omega) \leq n$. Let $E(\beta)$ be either $E(d)$ or a subtree of $E(d)$. By Lemmas 6.1.1 and 6.1.2, $intbits(\beta) \leq n\,2^{T(n)}$. Therefore, any leaf in $E(d)$ can be specified by a pointer of length $O(T(n) + \log n) = O(T(n))$. (The tree has $2T(n)$ levels, and we need space $O(T(n))$ to specify the branch at each level.) □

We describe here an efficient simulation of a PRAM[↑, ↓] by a basic PRAM. Let S be a PRAM[↑,↓] that uses $T(n)$ time and $P(n)$ processors. Let S' be a PRAM[↑,↓] that uses only short addresses and simulates S according to the Associative Memory Lemma. Thus, S' uses $O(P^2(n)T(n))$ processors, $O(T(n))$ time, and only addresses in 0, 1, ..., $O(P(n)T(n))$. Let q be a constant such that each processor in S' uses only q registers.

For numbers generated by S (and therefore S'), the depth of the encoding is at most $2T(n)$, and every internal node has at most $n\,2^{T(n)}$ children. Therefore, the encoding may have up to $4^{T^2(n)}$ nodes.

We construct a PRAM Z that simulates S via S' in $O(T^2(n))$ time, using $O(P^2(n)\,T(n)\,4^{T^2(n)})$ processors. For ease of description, we allow Z to have $q+7$ separate shared memories, mem_0, ..., mem_{q+6}, which can be interleaved. This entails no loss of generality and only a constant factor time loss (Lemma 3.1).

Initialization. Z partitions mem_0 into $O(P(n)T(n))$ *blocks* of $4^{T^2(n)}$ cells each. This partitioning allots one block per cell accessed by S', where each block comprises one cell per node of the encoding tree. Z partitions each of mem_1, ..., mem_q into $O(P^2(n)T(n))$ blocks. This allots one block per processor of S' and one memory per local register used by a

processor. (See Figure 6.2.) Let $B_i(m)$ denote the mth block of mem_i. Throughout the simulation, $B_0(j)$ contains E($con(j)$), and $B_i(m)$, $1 \le i \le q$, contains E($rcon_m(i)$) of S'.

Z activates $O(P^2(n)\,T(n))$ *primary* processors, one for each processor used by S'. In mem_{q+1}, these processors construct an address table. The jth entry of this table is $j \cdot 4^{T^2(n)}$, the address of the first cell of the jth block in every memory. The maximum address is $O(P(n)T(n)4^{T^2(n)})$, so this address (and the entire table) is computed in $O(T^2(n))$ time.

Each primary processor now deploys $4^{T^2(n)}$ *secondary* processors, one for each cell in a block, in $O(T^2(n))$ time. To implement a broadcast in constant time, each primary processor P_m uses $c_{q+2}(m)$ as a communication cell. When the secondary processors are not otherwise occupied, they concurrently read this cell at each time step, waiting for a signal from the primary processor to indicate their next tasks.

Consider a complete d-ary tree Λ with depth $2T(n)$. We number the nodes of Λ, starting with the root as node 1, in the order of a right-to-left breadth-first traversal. Node number j has children numbered $dj-(d-2)$, ..., dj, $dj+1$; its parent is numbered $\lfloor (j+(d-2)) / d \rfloor$.

mem_0	shared memory
mem_1, ..., mem_q	local memories
mem_{q+1}	address table
mem_{q+2}	communication
mem_{q+3}	rightmost child
mem_{q+4}	parent
mem_{q+5}	rightmost sibling
mem_{q+6}	two's complement representation of cell contents

Figure 6.2. Shared memories of Z

We view a block as a linear array storing Λ with $d = 4^{T(n)}$. Node numbers correspond to locations in the array. Let *node* (j) denote the node whose number is j. Let *num* (α) denote the node number of node α. Let $p(\alpha)$ denote the parent of node α; let *rc* (α) denote the rightmost child of node α. For each primary processor, the jth secondary processor, $1 \le j \le 4^{T^2(n)}$, handles *node* (j). Let *proc* (α) denote the secondary processor assigned to node α.

Each encoding is a subtree of Λ because some encoding nodes may have fewer than $4^{T(n)}$ children. Let *lc* (α) denote the leftmost nonempty child of node α. When a primary processor and its secondary processors update E(*con* (i)) or E($rcon_g(i)$), they also update *num* (*lc* (α)) for every node α. Let *right* (α) denote *num* (α) − *num* (*rc* $(p(\alpha))$). That is, *right* (α) denotes which child α is of $p(\alpha)$, counting from the right. Similarly, let *left* (α) denote *num* (α) − *num* (*lc* $(p(\alpha))$). That is, *left* (α) denotes which child α is of $p(\alpha)$, counting from the left.

Using mem_{q+2} for communication with primary processor P_h, corresponding to processor P_0 of S', *proc* (*node* (j)), $1 \le j \le 4^{T^2(n)}$, computes *num* (*rc* (*node* (j))) in $O(T(n))$ time and writes it in $c_{q+3}(j)$. Next, *proc* (*node* (j)) determines *num* (*p* (*node* (j))) and writes this number in mem_{q+4}. It writes j in c_{q+4}(*num* (*rc* (*node* (j)))). Note that j is *num* (*p* (*rc* (*node* (j)))). On the kth cycle, each secondary processor that has *num* (*p* (*node* (j))) (that is, $con_{q+4}(j) \neq 0$) writes that number 2^k cells away (in $mem_{q+4}(j+2^k)$), if that cell is empty. After Z repeats this procedure $T(n)-1$ times, $con_{q+4}(j)$ = *num* (*p* (*node* (j))), for all j. In this procedure, the processors also write *num* (*rc* (*p* (*node* (j)))) in $mem_{q+5}(j)$. Then the processor for each node j can compute *right* (j).

All the addresses of cells accessed by S' can be constructed using only addition and subtraction. In order to quickly perform indirect addressing, Z generates all cell and register contents in standard two's complement representation, except for results of shifts. If the value v in a register or a shared memory cell is the result of a shift, then S' will not use v as an address, and S' will use no other value computed from v as an address.

Recall that register addresses are in the range $0, \cdots, q-1$. The two's complement representation of local register $r_g(i)$ of S', if $rcon_g(i)$ is constructed without shifts, is stored in $c_{q+6}(g(q+1)+i)$. The two's complement representation of shared memory cell $c(j)$ of S', if $con(j)$ is constructed without shifts, is stored in $c_{q+6}((j+1)(q+1))$.

As the final initialization step, Z converts the input to the MIB encoding, writing the encoding into $B_0(0)$. Z writes the input integer in $c_{q+6}(q+1)$.

Simulation. In a general step of processor P_g of S', P_g executes instruction *instr*. Assume for now that *instr* has the form $r(i) \leftarrow r(j)\ \mathrm{O}\ r(k)$. To simulate this step, the corresponding primary processor P_m of Z and its secondary processors perform four tasks:

Task 1. If O is not a shift, then perform O on $con_{q+6}(g(q+1)+j)$ and $con_{q+6}(g(q+1)+k)$, writing the result in $con_{q+6}(g(q+1)+i)$.

Task 2. Merge the first level of the encodings $\mathrm{E}(rcon_g(j))$ and $\mathrm{E}(rcon_g(k))$.

Task 3. Determine where the interesting bits of $\mathrm{E}(rcon_g(i))$ occur in the merged encodings and compute their marks.

Task 4. Compress these marked interesting bits into the proper structure.

Z uses the procedures *MERGE* in Task 2 and *COMPRESS* in Task 4. Depending on the operation O, Z may also use the procedures *BOOL* and *ADD* in Task 3. These procedures are described below.

Procedures *MERGE*, *COMPRESS*, *BOOL*, and *ADD* call procedure *COMPARE*, which we now specify. Let j and k be nonnegative integers, and let ψ_1 and ψ_2 be encoding pointers. If $m = \lambda$, the empty string, then *COMPARE* $(j, \psi_1, k, \psi_2, m)$ compares the value of subtree $E(con(j)).\psi_1$ with the value of subtree $E(con(k)).\psi_2$. *COMPARE* returns "equal" if $val(E(con(j)).\psi_1) = val(E(con(k)).\psi_2)$, "greater" if $val(E(con(j)).\psi_1) > val(E(con(k)).\psi_2)$, or "less" if $val(E(con(j)).\psi_1) < val(E(con(k)).\psi_2)$. Similarly, if $m \neq \lambda$, then *COMPARE* compares the value of subtree $E(rcon_m(j)).\psi_1$ with the value of subtree $E(rcon_m(k)).\psi_2$. *COMPARE* returns "equal" if $val(E(rcon_m(j)).\psi_1) = val(E(rcon_m(k)).\psi_2)$, "greater" if $val(E(rcon_m(j)).\psi_1) > val(E(rcon_m(k)).\psi_2)$, or "less" if $val(E(rcon_m(j)).\psi_1) < val(E(rcon_m(k)).\psi_2)$.

Suppose $m = \lambda$; the case $m \neq \lambda$ is similar. For each node α in the first level of $E(con(j)).\psi_1$ simultaneously, *proc* (α) determines *left* (α). Then *proc* (α) computes *num* (β) such that node β is in the first level of $E(con(k)).\psi_2$ and *left* (β) = *left* (α) by reading *num* $(lc(E(con(k)).\psi_2))$. Next, *proc* (α) recursively compares the values of the subtrees rooted at α and β. If the interesting bit whose location is specified by *val* (α) is the end of a constant interval of 1's (0's), then *proc* (α) writes in the *num* (α)th cell in $B_{q+2}(g)$ which subtree has the greater (lesser) value, but writes nothing if the subtrees have equal value. By the concurrent write rules, the *num* (α)th cell in $B_{q+2}(g)$ either holds the name of the cell (j or k) whose subtree has greater value or holds nothing if they are equal.

Computing these node numbers takes constant time. *COMPARE* is recursive in the depth of the encoding, taking constant time at each level. Consequently, *COMPARE* $(j, \psi_1, k, \psi_2, m)$ takes $O(T(n))$ time.

In Task 2, Z merges the first level of the encodings $E(rcon_g(j))$ and $E(rcon_g(k))$. Z does this to compare the positions of interesting bits in $rcon_g(j)$ and $rcon_g(k)$. This comparison is necessary to determine the positions of the interesting bits in $rcon_g(i)$.

The subtrees rooted at the first level of $E(d)$ form a list sorted in increasing order by their values. The beginning of the list corresponds to the rightmost child of the root. $MERGE(j, k, i)$ returns, in $B_i(g)$, the list resulting from merging the first levels of $E(rcon_g(j))$ and $E(rcon_g(k))$. The list contains up to $O(2^{T(n)})$ subtrees, each of which is a subtree of $E(rcon_g(j))$ or $E(rcon_g(k))$. Each subtree in the merged list retains indications of whether it is from j or k, whether it is the end of a constant interval of 0's or 1's, and its (singleton) mark.

In parallel, Z compares $val(\alpha)$, for each subtree rooted at α in the first level of $E(rcon_g(j))$, with $val(\beta)$, for each subtree rooted at β in the first level of $E(rcon_g(k))$. If $val(\alpha) > val(\beta)$, then $proc(\alpha)$ writes $right(\beta)$ into the $num(\alpha)$th cell of $B_{q+2}(g)$. By the concurrent write rules, this cell contains the largest $right(\beta)$ for which $val(\alpha) > val(\beta)$. Call this value $max(\beta)$. For each α, $right(\alpha) + max(\beta)$ specifies the position of the subtree rooted at α in the merged list. Next, for each β, if $val(\beta) \geq val(\alpha)$, then $proc(\beta)$ writes $right(\alpha)$ into the $num(\beta)$th cell of $B_{q+2}(g)$. Call this value $max(\alpha)$. For each β, $right(\beta) + max(\alpha)$ specifies the position of the subtree rooted at β in the merged list. A comparison takes $O(T(n))$ time, so Z performs a $MERGE$ in $O(T(n))$ time. (Note: Each subtree in the merged list also indicates whether its value is equal to that of the succeeding subtree in the list.)

We introduce one more procedure before describing the computation of the interesting bits of $rcon_g(i)$. Let $I(d)$ denote the MIB encoding of d without the marks.

PLUS_ONE (k, ψ_1, i, ψ_2) writes $I(val(E(rcon_g(k)).\psi_1) + 1)$ in the location set aside for subtree ψ_2 in $B_i(g)$. That is, given $E(d)$, for d an integer, *PLUS_ONE* writes $I(d+1)$. *PLUS_ONE* does not write singleton marks. Z uses *PLUS_ONE* to generate $I(d+1)$ to test for equality with $E(x)$, x an integer. The processors ignore marks to interpret $E(x)$ as $I(x)$. At most, the two rightmost interesting bits of $d+1$ will be different from those of d. We have four cases to consider:

(a) d starts with a 0, the 0 is a singleton,

(b) d starts with a 0, the 0 is not a singleton,

(c) d starts with a 1, the first 0 is a singleton, and

(d) d starts with a 1, the first 0 is not a singleton.

In every case, Z complements the start bit. In case (a), Z deletes the first interesting bit in $E(d)$. In case (b), Z adds a new interesting bit at location 0. In case (c), Z deletes the second interesting bit. In case (d), let us suppose the first interesting bit is at location f. Z deletes this interesting bit and creates a new interesting bit at location $f+1$ by recursively calling *PLUS_ONE*. (Naturally, when Z adds or deletes one interesting bit, it shifts the subtrees encoding the locations of the other interesting bits one position left or right, as necessary. In a block, the associated processors copy their respective subtrees in constant time.)

PLUS_ONE is recursive with depth $T(n)$, the depth of the encodings. *PLUS_ONE* uses constant time at each level, so $O(T(n))$ time overall.

We now are ready to describe how Z accomplishes Task 3. Assume without loss of generality that i, j, and k are different. Z's actions in Task 3 depend on the operation $\bigcirc$ in *instr*. Define an *interval-pair* to be the intersection of a constant interval in $rcon_g(j)$ and a

constant interval in $rcon_g(k)$. For example, three interval-pairs, denoted a, b, and c, are shown below.

	$c\ c\ b\ b\ b\ a$
$rcon_g(j)$	1 1 0 0 0 1
$rcon_g(k)$	0 0 1 1 1 1

The *interval-pair length* of interval-pair a is 1, of interval-pair b is 3, and of interval-pair c is 2.

ZERO_ONE $(j,\ k,\ i)$ takes as input the merged list from $E(rcon_g(j))$ and $E(rcon_g(k))$ in $B_i(g)$ and returns as output an indication for each subtree in the list from $E(rcon_g(j))$ (respectively, $E(rcon_g(k))$) whether $rcon_g(k)$ (respectively, $rcon_g(j)$) is in a constant interval of 0's or 1's at the location specified by the value of the subtree. The secondary processors handling the merged list act as a binary computation tree to pass along the desired information in $O(T(n))$ time.

IP_LENGTH $(j,\ k,\ i)$ takes as input the merged list from $E(rcon_g(j))$ and $E(rcon_g(k))$ in $B_i(g)$ and returns as output an indication for each subtree in the list whether the interval pair ending at the location specified by the value of the subtree has length 1 or greater than 1. To perform this computation, Z calls *PLUS_ONE* $(i,\ \psi_1,\ i',\ \psi_1)$ in parallel for each subtree in the list, where ψ_1 is the location of the subtree in the list and i' refers to $B_{q+2}(i)$. Suppose the subtree encodes the integer d. Then, in parallel, Z tests $I(d+1)$ for equality with the succeeding subtree in the list. If they have equal value, then the interval-pair length is 1; otherwise, the interval-pair length is greater than 1.

If *instr* is $r(i) \leftarrow r(j) \vee r(k)$, then Z calls *BOOL* $(j,\ k,\ i,\ \vee)$. *BOOL* $(j,\ k,\ i,\ \vee)$ writes $E(rcon_g(j) \vee rcon_g(k))$ in $B_i(g)$. Assume that we have the merged encodings of $E(rcon_g(j))$ and $E(rcon_g(k))$ in $B_i(g)$. Z must compute the interesting bits and their marks. Z performs

two preliminary steps:

(1) for each subtree in the list from E($rcon_g(j)$) (respectively, E($rcon_g(k)$)) determine whether $rcon_g(k)$ (respectively, $rcon_g(j)$) is in a constant interval of 0's or 1's at the location specified by the value of the subtree and

(2) determine whether the interval-pair length is 1.

Z calls *ZERO_ONE* (j, k, i) and *IP_LENGTH* (j, k, i) to perform (1) and (2), respectively. A subtree is *interesting* if its value is the location of an interesting bit of $rcon_g(i)$; otherwise, the subtree is *boring*. Following the rules in Appendix A, the processors associated with each subtree tag it as "interesting" or "boring." It remains for Z to compute the marks of the interesting bits. For each interesting subtree, the processor associated with the root determines the following. If the interval-pair has length 1 and the preceding subtree in the list is a nonblank, then mark it s; otherwise, mark it m. The entire procedure takes time $O(T(n))$. Other Boolean instructions are handled similarly.

If *instr* is $r(i) \leftarrow r(j) + r(k)$, then Z calls *ADD* (j, λ, k, λ, i, λ).

ADD (j, ψ_1, k, ψ_2, i, ψ_3) writes E(val(E($rcon_g(j)$).ψ_1) + val(E($rcon_g(k)$).ψ_2)) in the location set aside for subtree ψ_3 in $B_i(g)$. Again, assume that we have the list of merged first level subtrees of E($rcon_g(j)$) and E($rcon_g(k)$) in $B_i(g)$. To accomplish Task 3, Z must test four conditions at the bit location specified by the value of the subtree:

(a) whether the $rcon_g(j)$ and $rcon_g(k)$ pairs are both in constant intervals of 0's, both in 1's, or one in 0's and one in 1's,

(b) whether there is a carry-in to the interval-pair,

(c) whether $rcon_g(i)$ is in a constant interval of 0's or 1's prior to the start of the interval-pair, and

(d) whether the interval-pair length is 1 or greater than 1.

Z calls *ZERO_ONE* and *IP_LENGTH* to test conditions (a) and (d) in time $O(T(n))$. For each subtree α in the list, *proc* (α) does the following. To test condition (b), *proc* (α) tests condition (a) at the preceding subtree in the list. If both $rcon_g(j)$ and $rcon_g(k)$ are in 0's, then there is no carry-in; if both are in 1's, then there is a carry-in. If one is in 0's and the other is in 1's, then the carry-in depends on the carry-in to the preceding interval-pair. To propagate this information in time $O(T(n))$, processors again act as a binary computation tree. To test condition (c), *proc* (α) determines whether $rcon_g(i)$ is in a constant interval of 0's or 1's at position *val* (α) using the other three conditions, then passes this information to *proc* (*num* (α)+1). The processors act as a binary computation tree of height $O(T(n))$ in testing all four conditions. Thus, all four conditions can be tested in $O(T(n))$ time.

Following the rules in Appendix B, for each subtree α in the list, *proc* (α) determines whether subtree α encodes the position of an interesting bit of $rcon_g(i)$. In doing so, *proc* (α) may call the procedure *PLUS_ONE*. This leads to an overall time of $O(T(n))$ to perform an addition.

If *instr* is $r(i) \leftarrow r(j) - r(k)$, then Z computes $E(rcon_g(j)+1)$ in $B_i(g)$ by a call to *ADD*, then calls *ADD* $(i, \lambda, k', \lambda, i, \lambda)$, where k' indicates that the start bit of $E(rcon_g(k))$ is complemented (thus adding $rcon_g(j)$ and the two's complement of $rcon_g(k)$). This takes $O(T(n))$ time.

If $rcon_g(k) < 0$ and *instr* is $r(i) \leftarrow r(j) \uparrow r(k)$, then Z treats *instr* as $r(i) \leftarrow r(j) \downarrow r(k)$, substituting $|rcon_g(k)|$ for $rcon_g(k)$. Similarly, if $rcon_g(k) < 0$ and *instr* is $r(i) \leftarrow r(j) \downarrow r(k)$, then Z treats *instr* as $r(i) \leftarrow r(j) \uparrow r(k)$, substituting $|rcon_g(k)|$ for $rcon_g(k)$. Thus, for both shift instructions, we shall assume $rcon_g(k) \geq 0$.

If *instr* is $r(i) \leftarrow r(j) \uparrow r(k)$, then the dth interesting bit of $rcon_g(i)$ is in the position specified by the sum of $rcon_g(k)$ and the position of the dth (if the least significant bit of $rcon_g(j)$ is 0) or $d+1$st (if the least significant bit of $rcon_g(j)$ is 1 and $rcon_g(k) \neq 0$) interesting bit of $rcon_g(j)$. Z adds $rcon_g(k)$ to the value of each subtree from $rcon_g(j)$. Marks stay the same, except perhaps for the first interesting bit of $rcon_g(j)$: if it has mark s and $rcon_g(k) = 0$, then Z marks it s; otherwise, Z marks it m. This procedure takes $O(T(n))$ time, the time to perform *ADD*.

If *instr* is $r(i) \leftarrow r(j) \downarrow r(k)$, then Z subtracts $rcon_g(k)$ from the value of each first level subtree of $rcon_g(j)$. Z tags as boring those subtrees for which this difference is negative. For the others, this difference is the location of an interesting bit in $rcon_g(i)$. Let γ denote the subtree whose value specifies the location of the first interesting bit in $rcon_g(i)$. Marks stay the same, except perhaps for γ: if $val(\gamma) = 0$, then Z marks it s; otherwise, Z marks it m. The start bit of $rcon_g(i)$ depends on whether $rcon_g(j)$ is in a constant interval of 0's or 1's at the location specified by the subtree that became γ. This procedure takes time $O(T(n))$, the time to perform *ADD*.

Z accomplishes Task 4 by calling *COMPRESS*. *COMPRESS*(i) takes the contents of block i, which implicitly stores a tree in which some subtrees rooted at the first level are tagged to be deleted (boring), and rewrites the tree without the boring subtrees. The secondary processors act as a binary computation tree so that the processors associated with the root of each interesting (that is, not boring) first level subtree can determine the number of interesting subtrees to the right in time $O(T(n))$. This number specifies the location of the subtree in the compressed tree. Then Z copies each subtree into the appropriate location and writes zeroes in the unused locations. Overall, *COMPRESS*(i) takes $O(T(n))$ time.

Now let us consider instructions *instr* executed by processor P_g of S' where *instr* has a form other than $r(i) \leftarrow r(j)\ O\ r(k)$.

If *instr* is $r(i) \leftarrow c(r(j))$, then P_m reads $y = con_{q+6}(g(q+1)+j)$, the two's complement representation of $rcon_g(j)$. P_m and its secondary processors then copy $B_0(y)$ into $B_j(g)$. P_m also writes $con_{q+6}((y+1)(q+1))$ in $c_{q+6}(g(q+1)+j)$. We handle instruction *instr* of the form $r(i) \leftarrow r(r(j))$ similarly.

If *instr* is $c(r(i)) \leftarrow r(j)$, then P_m reads $y = con_{q+6}(g(q+1)+i)$, the two's complement representation of $rcon_g(i)$. P_m and its secondary processors then copy $B_j(g)$ into $B_0(j)$. P_m also writes $con_{q+6}(g(q+1)+j)$ in $c_{q+6}((y+1)(q+1))$. We handle instruction *instr* of the form $r(r(i)) \leftarrow r(j)$ similarly.

If processors P_f and P_g of S' wish to simultaneously write $c(j)$, then the corresponding processors P_l and P_m of Z will simultaneously attempt to write $B_0(j)$. If $f < g$, then $l < m$, and all secondary processors of P_l are numbered less than all secondary processors of P_m. Thus, in S', P_f succeeds in its write, and in Z, P_l and its secondary processors succeed in their writes.

At most $4^{T^2(n)}$ processors simultaneously read or write a cell in the simulation described above. Note that simultaneous reads and writes occur in two ways: (a) $O(4^{T^2(n)})$ processors simultaneously read or write a cell at one step of an $O(T(n))$ step procedure, and (b) $O(2^{T(n)})$ processors simultaneously read or write at each step of an $O(T(n))$ step procedure. As a result, if we wish to restrict the number of simultaneous reads and writes, we can revise the simulation with no time loss such that all simultaneous reads and writes of form (a) are modified to form (b).

Theorem 6.1. For all $T(n) \geq \log n$, $PRAM\,[\uparrow,\downarrow]\text{-}TIME\,(T(n)) \subseteq PRAM\text{-}TIME\,(T^2(n))$.

Proof. In the simulation given above, Z takes $O(T(n))$ time per step of S' to merge two encodings, compute new marked interesting bits, and compress the list into the proper MIB form. S' simulates S in $O(T(n))$ time. Hence, Z takes $O(T^2(n))$ time to simulate S via S'. □

Corollary 6.1.1. $PRAM\,[\uparrow,\downarrow]\text{-}POLYLOGTIME = PRAM\text{-}POLYLOGTIME$.

6.2. Simulations of PRAM[↑,↓] by Circuits and Turing Machine

We now describe simulations of a PRAM[↑,↓] by a log-space uniform family of unbounded fan-in circuits, a log-space uniform family of bounded fan-in circuits, and a Turing machine.

Lemma 6.2.1. For each n, every language recognized by a PRAM[↑,↓] S in time $T(n)$ with $P(n)$ processors can be recognized by a log-space uniform unbounded fan-in circuit C_n of depth $O(T^2(n))$ and size $O(P^4(n)T^4(n)16^{T^2(n)})$.

Proof. Let Z be the PRAM described in Theorem 6.1, simulating S in $O(T^2(n))$ time with $O(P^2(n)T(n)4^{T^2(n)})$ processors. Fix an input length n. We construct an unbounded fan-in circuit C_n that simulates Z by the construction given by Theorem 2.1. □

Lemma 6.2.2. For each n, every language recognized by a PRAM[↑,↓] S in time $T(n)$ with $P(n)$ processors can be recognized by a log-space uniform unbounded fan-in circuit UC_n of depth $O(T^2(n))$, size $O(P^4(n)T^4(n)16^{T^2(n)})$, and maximum fan-in $O(4^{T(n)}T^2(n))$.

Proof. Fix an input length n. We construct UC_n from C_n of Lemma 6.2.1. We reduce the fan-in in the portions of the circuit that simulate updates in the shared memory of Z. The

circuit described in Theorem 2.1 allows all processors to attempt to simultaneously write the same cell. This does not occur in Z. During the execution of each procedure of Z, either $4^{T^2(n)}$ secondary processors concurrently write the same cell once or $4^{T(n)}$ secondary processors concurrently write the same cell at each of $O(T(n))$ levels of recursion. Thus, we can modify Z such that at most $4^{T(n)}$ processors attempt to write the same cell at each time step, keeping the time for each procedure at $O(T(n))$. By the construction given in Theorem 2.1, this leads to a maximum fan-in for any gate in UC_n of $O(4^{T(n)}T^2(n))$ if $T(n) \geq n$ or $O(4^{T(n)}T(n)n)$ if $T(n) < n$. The circuit remains uniform after modifications to Z because the processors concurrently writing are all secondary processors belonging to the same primary processor. UC_n has depth $O(T^2(n))$ and size $O(P^2(n)T^4(n)4^{T^2(n)}(T(n) + P^2(n)4^{T^2(n)})) = O(P^4(n)T^4(n)16^{T^2(n)})$. □

Lemma 6.2.3. For each n, every language recognized by a PRAM[↑,↓] S in time $T(n)$ with $P(n)$ processors can be recognized by a log-space uniform bounded fan-in circuit BC_n of depth $O(T^3(n))$ and size $O(P^4(n)T^4(n)16^{T^2(n)})$.

Proof. Fix an input length n. Let UC_n be the unbounded fan-in circuit described in Lemma 6.2.2 that simulates S. The gates of UC_n have maximum fan-in of $O(2^{T(n)}T^2(n))$ if $T(n) \geq n$ or $O(2^{T(n)}T(n)n)$ if $T(n) < n$. We construct the bounded fan-in circuit BC_n by replacing each gate of UC_n with fan-in f by a tree of gates of depth $\log f$. Since every $f = O(2^{T(n)}(T^2(n) + nT(n))$, and $T(n) \geq \log n$, BC_n can simulate each gate of UC_n in depth $O(T(n))$. Since UC_n has depth $O((T^2(n))$ by Lemma 6.2.2, BC_n has depth $O(T^3(n))$. □

Theorem 6.2. For all $T(n) \geq \log n$, $PRAM[\uparrow,\downarrow]\text{-}TIME(T(n)) \subseteq DSPACE(T^3(n))$.

Proof. Theorem 6.2 follows from Lemma 6.2.3 and Borodin's (1977) result that a bounded fan-in circuit of depth $D(n)$ can be simulated in space $O(D(n))$ on a Turing machine. □

Theorem 6.2 and a fundamental result of Fortune and Wyllie (1978)

$$DSPACE(T(n)) \subseteq PRAM\text{-}TIME(T(n)) \text{ for all } T(n) \geq \log n$$

together imply that $PRAM[\uparrow,\downarrow]\text{-}TIME(T(n)) \subseteq PRAM\text{-}TIME(T^3(n))$. The direct simulation of Theorem 6.1 is more efficient.

Theorem 6.1 and the other fundamental result of Fortune and Wyllie

$$PRAM\text{-}TIME(T(n)) \subseteq DSPACE(T^2(n)) \text{ for all } T(n) \geq \log n$$

together imply that $PRAM[\uparrow,\downarrow]\text{-}TIME(T(n)) \subseteq DSPACE(T^4(n))$. The $O(T^3(n))$ space simulation of Theorem 6.2 is more efficient.

Corollary 6.2.1. $PRAM[\uparrow,\downarrow]\text{-}PTIME = PSPACE$.

6.3. Direct Simulation of PRAM[↑,↓] by Turing Machine

In the previous section, we indirectly simulated a PRAM[↑,↓] by a Turing machine. Here, we present a direct simulation that achieves the same space bound.

We use the *interesting bit (IB) encoding* of Simon (1977). Let d be an integer, and let $w = len(d)$. Let $b_{w-1} \cdots b_0$ be the w-bit two's complement representation of d. We define the interesting bit encoding as

$$I(0) = 0,$$

$$I(01) = 1,$$

$$I(d) = (I(a_k), \ldots, I(a_2), I(a_1); r),$$

where d is an integer, a_j is the position of the jth interesting bit of d, and r is the value (0 or 1) of the rightmost bit of d.

For example, I(01100) = (I(011), I(01); 0) = ((I(01); 1), 1; 0) = ((1; 1), 1; 0). For all d, define $val(I(d)) = d$ and $\overline{val}(I(d)) = \bar{d}$, that is, $\overline{val}$ returns the complement of the two's complement

representation of d. Observe that the IB encoding is simply the MIB encoding without the marks.

We simulate a PRAM[$\uparrow,\downarrow$] S running in time $T(n)$ by a TM running in space polynomial in $T(n)$ by writing only pointers into the encodings of cell contents. This manipulation of pointers to individual symbols in the encoding is similar to the manipulation of individual bits in the simulation of a PRAM[*] by a TM in Section 4.3. During the computation of S on any input of length n, for every $c(j)$, the length of I($rcon(j)$) is at most exponential in $T^2(n)$, and pointers have length at most $T^2(n)$ (Lemma 6.1.3). We invoke the Associative Memory Lemma to construct a PRAM[$\uparrow,\downarrow$] S' that simulates S using only short addresses.

Simulation. We now describe the simulation of a time-bounded PRAM[$\uparrow,\downarrow$] by a space-bounded TM. Let S be a PRAM[$\uparrow,\downarrow$] that uses $T(n)$ time and $P(n)$ processors. Let S' be a PRAM[$\uparrow,\downarrow$] that uses only short addresses and simulates S according to the Associative Memory Lemma. Thus, S' uses $O(P^2(n)T(n))$ processors, $O(T(n))$ time, and only addresses in 0, 1, ..., $O(P(n)T(n))$.

We construct a TM M that simulates S via S' in $T^3(n)$ space. M uses four mutually recursive procedures: *PCOUNTER*, *COMPARE*, *SYMBOL*, and *ADD*. In the following, α (and β) may have any of the following forms: (i) $\#d$, where d is a constant, (ii) j, where j is a register address and I(α') is I($rcon_m(j)$), (iii) $j.\phi$, where ϕ is a pointer and I(α') is I($rcon_m(j)$).ϕ, (iv) $1 + j.\phi$, where I(α') is I($1 + val$(I($rcon_m(j)$).ϕ)), or (v) $1 + \overline{j.\phi}$, where I(α') is I($1 + \overline{val}$ (I($rcon_m(j)$).ϕ)). During the simulation, every $\#d$, j, and $j.\phi$ parameter can be written in $O(T^2(n))$ space.

PCOUNTER (m, t) returns the contents of the program counter of P_m at time t. To determine whether S' accepts input ω, M executes *PCOUNTER* to check whether P_0 halts with its program counter on an *ACCEPT* instruction by time $O(T(n))$. Let p be the value returned by *PCOUNTER* (m, t). *PCOUNTER* (m, t) depends on r, the value returned by *PCOUNTER* $(m, t-1)$. If r indicates that P_m was not active at time $t-1$, then *PCOUNTER* determines whether $P_{\lfloor m/2 \rfloor}$ activated P_m at time $t-1$ with a *FORK* instruction by calling *PCOUNTER* $(\lfloor m/2 \rfloor, t-1)$. If $P_{\lfloor m/2 \rfloor}$ executed *FORK label* 1, *label* 2, then if m is even, $p =$ *label* 1; otherwise, $p =$ *label* 2. If $P_{\lfloor m/2 \rfloor}$ did not execute *FORK*, then P_m is inactive at time t. If r indicates that P_m is active at time $t-1$ and step r is not a *CJUMP*, *REJECT*, or *ACCEPT* instruction, then $p = r+1$. If step r is *CJUMP* $r(i)$ *comp* $r(j)$, *label* 3, where *comp* is an integer comparison, then *PCOUNTER* repeatedly calls *VALUE* for time $t-1$ to compare $rcon_m(i)$ and $rcon_m(j)$. If the comparison is true, then $p =$ *label* 3; otherwise, $p = r+1$. If instruction r is an *ACCEPT* (*REJECT*), then $p = r$.

In the following procedures, if $m = \lambda$, then interpret α and β as referring to shared memory cells; if $m \neq \lambda$, then interpret α and β as referring to registers of P_m. We describe the case $m \neq \lambda$.

COMPARE $(\alpha, \psi_1, \beta, \psi_2, m, t)$ compares the value of subtree $I(\alpha').\psi_1$ at time t (of the computation of S') with the value of subtree $I(\beta').\psi_2$ at time t. *COMPARE* returns "equal" if $val(I(\alpha').\psi_1) = val(I(\beta').\psi_2)$, "greater" if $val(I(\alpha').\psi_1) > val(I(\beta').\psi_2)$, or "less" if $val(I(\alpha').\psi_1) < val(I(\beta').\psi_2)$. *COMPARE* recursively compares the subtrees of $I(\alpha').\psi_1$ and $I(\beta').\psi_2$ from right to left. *COMPARE* calls *SYMBOL* to determine whether the elements considered are subtrees or leaves, and, if they are leaves, to determine the symbol at the leaf. The interested reader is referred to Appendix C for the details of *COMPARE*.

SYMBOL (α, ψ, m, t) returns the symbol I(α').ψ if ψ points to a leaf of I(α'); otherwise, *SYMBOL* returns a signal that ψ points to a subtree of I(α').

Assume α has the form i. *SYMBOL* calls *PCOUNTER* to determine whether P_m wrote register $r_m(i)$ at time $t-1$. If P_m did not write $r_m(i)$, then *SYMBOL* returns *SYMBOL* (i, ψ, m, $t-1$). Otherwise, suppose P_m executed an instruction *instr* that wrote $r_m(i)$ at time $t-1$.

If *instr* was $r(i) \leftarrow d$, then this is a base case. If ψ points to a leaf of the encoding of the constant d, then *SYMBOL* returns $I(d).\psi$; otherwise, *SYMBOL* returns a signal that ψ points to a subtree.

If *instr* was $r(i) \leftarrow r(j)$, then *SYMBOL* returns *SYMBOL* (j, ψ, m, $t-1$). If $t = 0$, then *SYMBOL* (i, ψ, m, t) is a base case. M can determine I($rcon_m(i)$).ψ because $rcon_m(0)$ at time 0 is m; all other registers contain 0, and M has space to write the processor number in encoded form. (Note: If $m = \lambda$, then $t = 0$ is again a base case. At time 0, $con(0)$ is the input, all other cells contain 0, and M has space to write the input in encoded form.)

For Boolean operations, *SYMBOL* is straightforward. See Appendix D for details.

If *instr* was $r(i) \leftarrow r(j) + r(k)$, then *SYMBOL* returns *ADD* (j, k, ψ, m, $t-1$), which returns I($rcon_m(j) + rcon_m(k)$).ψ.

If *instr* was $r(i) \leftarrow r(j) - r(k)$, then *SYMBOL* returns *ADD* (j, $1+\bar{k}$, ψ, m, $t-1$). Recall that $1 + \overline{rcon_m(k)}$ is the two's complement of $rcon_m(k)$.

If $rcon_m(k) < 0$ and *instr* was $r(i) \leftarrow r(j) \uparrow r(k)$, then M treats *instr* as $r(i) \leftarrow r(j) \downarrow r(k)$, substituting $|rcon_m(k)|$ for $rcon_m(k)$. Similarly, if $rcon_m(k) < 0$ and *instr* was $r(i) \leftarrow r(j) \downarrow r(k)$, then M treats *instr* as $r(i) \leftarrow r(j) \uparrow r(k)$, substituting $|rcon_m(k)|$ for $rcon_m(k)$. Thus, for both shift instructions, we shall assume $rcon_g(k) \geq 0$.

If *instr* was $r(i) \leftarrow r(j) \uparrow r(k)$, then the vth interesting bit of $rcon_m(i)$ will be in the position specified by the sum of $rcon_m(k)$ and the position of the vth (if the rightmost bit of $rcon_m(j)$ is 0) or $v+1$st (if the rightmost bit of $rcon_m(j)$ is 1) interesting bit of $rcon_m(j)$. For $\psi = x_0 x_1 \cdots x_t$, let $FIRST(\psi) = x_0$ and $REST(\psi) = x_1 x_2 \cdots x_t$. *SYMBOL* returns either $ADD(j.FIRST(\psi), k, REST(\psi), m, t-1)$ or $ADD(j.(FIRST(\psi)-1), k, REST(\psi), m, t-1)$.

If *instr* was $r(i) \leftarrow r(j) \downarrow r(k)$, then if $rcon_m(k)$ is greater than the position of the leftmost interesting bit of $rcon_m(j)$, then *SYMBOL* returns 0; otherwise, the ith interesting bit of $rcon_m(i)$ will be the ith interesting bit of $rcon_m(j)$ such that the difference between the value of its position and $rcon_m(k)$ is nonnegative. The interesting bit of $rcon_m(i)$ will be in the position specified by the difference between the value of the position of this interesting bit of $rcon_m(j)$ and $rcon_m(k)$. *SYMBOL* calls *ADD* to find this information.

If *instr* used an indirect address, say $c(r(j))$, then M uses *SYMBOL* to get $I(rcon_m(j))$ one symbol at a time. Next, M decodes $I(rcon_m(j))$ to get $rcon_m(j)$. Since all addresses used by S' are $O(T(n))$ long, M has space to write $rcon_m(j)$. Now *SYMBOL* can directly access the desired cell. If the indirect address was $r(r(g))$, then M reads $rcon_m(g)$ one symbol at a time. Since each processor in S' uses only a constant number of registers, $len(rcon_m(g))$ is a constant, and M has space to write the address.

$ADD(\alpha, \beta, \psi, m, t)$ returns the symbol to which ψ points in $I(val(I(\alpha')) + val(I(\beta')))$ if ψ points to a leaf in the encoding of the sum; otherwise, *ADD* returns a signal that ψ points to a subtree of the encoding of the sum. *ADD* computes subtrees of $I(val(I(\alpha')) + val(I(\beta')))$ from right to left and based on cases depending on whether $val(I(\alpha'))$ and $val(I(\beta'))$ are in constant intervals of 0's or 1's and on the carry from the previous bit position. Note that this procedure also works when β is of the form $1+\bar{k}$. See Appendix E for details.

Theorem 6.3. For all $T(n) \geq \log n$, $PRAM[\uparrow,\downarrow]\text{-}TIME(T(n)) \subseteq DSPACE(T^3(n))$.

Proof. The simulation given above simulates a PRAM[$\uparrow,\downarrow$] S by a TM M.

For each invocation of *PCOUNTER*, *COMPARE*, *SYMBOL*, and *ADD*, M can write all variables and parameters in space $O(\log P(n)T(n))$, $O(T^2(n))$, $O(T^2(n))$, and $O(T^2(n))$, respectively. The depth of recursion of these procedures is at most $O(T(n))$. Since $P(n) \leq 2^{T(n)}$, M uses space $O(T^3(n))$ to simulate S. With linear space compression, M uses space $T^3(n)$. □

Chapter 7. Multiplication and Shift

We study the interaction of multiplication and shift instructions in this chapter. Combined, they can produce very long and complex numbers. The product of two integers with b and b' interesting bits can have bb' interesting bits. Thus with s interleaved shift and multiplication operations, a PRAM[*,↑] can build numbers with $2^{2^{O(s)}}$ interesting bits.

Let *NEXPTIME* be the class of languages accepted by a nondeterministic Turing machine in time $O(c^{poly(n)})$, where c is a constant and $poly(n)$ is a polynomial in n; let *EXPSPACE* be the class of languages accepted by a Turing machine in space $O(c^{poly(n)})$, c a constant. In Section 7.1, we prove $NEXPTIME \subseteq PRAM[*, \uparrow]\text{–}PTIME$, and in Section 7.2, we prove $PRAM[*, \uparrow, \downarrow]\text{–}PTIME \subseteq EXPSPACE$. We have previously shown that polynomial time on a PRAM[*] or a PRAM[↑,↓] is equal to *PSPACE* on a Turing machine. Thus, a PRAM with both multiplication and shift instructions may be more powerful, to within a polynomial in time, than a PRAM with either multiplication or shift alone, since it is believed that *NEXPTIME* properly contains *PSPACE*.

7.1. Simulation of TM by PRAM[*,↑]

We present here a simulation of a nondeterministic Turing machine (NTM) running in exponential time by a PRAM[*,↑] running in polynomial time. Our strengthening of the simulation of a Turing machine from *PSPACE* to *NEXPTIME* relies on an interaction between multiplication and shift operations. For an integer v, let $\#v$ denote its two's complement representation; the number of bits in the two's complement representation will be clear from the context. As previously noted, the shift operation is useful for making copies of strings in a single cell. Multiplication can make copies much more quickly. If v is

an integer such that 1's are spaced widely enough in $\#v$, then multiplying an integer u by an integer v produces an integer w such that $\#w$ contains a copy of $\#u$ for every 1 in $\#v$.

We simulate a multitape NTM; the input initially appears on one of the tapes. A *configuration* of an NTM Q comprises the contents of its tapes, the position of its tape heads, and the state of its finite-state control. Let *state* (σ) denote the state of Q in configuration σ. For two configurations σ and τ of Q, the relation $\sigma \vdash \tau$ holds if Q in configuration σ may, in one step, make a transition to configuration τ according to its transition rules. A transition from configuration σ to configuration τ is *valid* if $\sigma \vdash \tau$ holds. A *computation* of Q running for $T(n)$ steps is a sequence of $T(n) + 1$ configurations $C = \sigma_0\sigma_1 \cdots \sigma_{T(n)}$, where the ith configuration, for all $0 \le i \le T(n)$, describes Q after the ith step, $\sigma_i \vdash \sigma_{i+1}$, and σ_0 is the initial configuration of Q with input ω. Computation C is *accepting* if *state* $(\sigma_{T(n)})$ is an accepting state.

A *neighborhood* in a configuration comprises the contents of two adjacent tape squares, one of which a tape head is reading, the location of the tape head on one of the squares, and the state of the TM. Let $N_{il}(\sigma)$ $(N_{ir}(\sigma))$ denote the neighborhood in configuration σ in which the tape head of the ith tape is on the left (right) square. For a one-tape NTM, we simply use $N_l(\sigma)$ and $N_r(\sigma)$.

We assume that every TM worktape is one-way infinite to the left. We make this assumption in order to simplify the relationship between a TM configuration and an encoding of that configuration built on a PRAM[*,↑], since the PRAM[*,↑] builds numbers that increase from left to right.

Let us outline the simulation. The simulating PRAM[*,↑], *MS*, generates in a single cell a description of all possible sequences of configurations by a time bounded NTM. *MS*

then tests each of these configuration sequences to determine whether at least one of them is an accepting computation of the NTM on the given input. If so, then *MS* accepts the input; otherwise, *MS* rejects. Multiplication and shift interact to quickly generate the description of all possible computations.

• *Oblivious motion*

To test the validity of a possible computation, *MS* must be able to easily locate the tape head in each configuration. For this reason, we have *MS* simulate an *oblivious Turing machine,* where the head motion is regular and does not depend on the input or tape contents. *NEXPTIME* can be characterized by oblivious simple NTMs working in exponential time.

Let Q be an NTM running in time $T(n)$ with a constant number of worktapes, one read-write head per tape, and with the input string ω initially written on one worktape. Since Q runs for $T(n)$ steps, it uses at most $T(n)$ space on each tape. We construct an oblivious Turing machine Q' that simulates Q. The simulator Q' has a single worktape $T(n) + 2$ squares long, with a special endmarker at each end. Q' will be a *sweeping* NTM. Its head moves one square at every step and does not change direction except at the endmarkers, so the head motion is a sequence of one-way sweeps back and forth across the worktape. The tape head of Q' halts to accept or reject the input only at one of the endmarkers. We partition the set of states of Q' into two sets, R and L. R denotes the set of states ϕ in which the tape head moves right, unless the symbol being read is the right endmarker. L denotes the set of states ϕ in which the tape head moves left, unless the symbol being read is the left endmarker.

Lemma 7.1.1. Q' simulates Q in $O(T^2(n))$ time.

Proof. We describe the simulation of a general step of Q by Q'. We describe the simulation for Q with a single worktape. To generalize the simulation to the case where Q has multiple tapes, Q' uses multiple tracks, one per tape of Q. Suppose Q' is in a right sweep; that is, Q' is in a state $\phi_{Q'} \in R$. The case where Q' is in a left sweep is handled similarly.

Suppose the simulated machine Q at this time is in state ϕ_Q and that Q writes a symbol s on the tape square currently being read, nondeterministically selects a next state χ_Q, and moves to the right. The simulating oblivious sweeping NTM Q' writes s, selects corresponding next state $\chi_{Q'}$, and moves to the right. This completes the simulation of this step of Q.

Suppose instead when Q is in state ϕ_Q that Q writes s, nondeterministically selects next state ψ_Q, and moves to the left. In this case, Q' writes symbol s^* as a marker for this position on the tape, selects state $\psi_{RQ'} \in R$, and moves to the right. On successive steps, Q' continues to sweep to the right in state $\psi_{RQ'}$ without writing until it reaches the right endmarker. Q' then enters state $\psi_{LQ'} \in L$ and begins a left sweep. Q' sweeps left in state $\psi_{LQ'}$ without writing until it reads symbol s^*. At this point, Q' writes symbol s, enters state $\psi_Q \in L$, and moves to the left. Now Q' is ready to simulate the next move of Q. This takes $O(T(n))$ steps.

If the tape head of Q remains stationary, then Q' simulates this step similarly to a left head motion in $O(T(n))$ steps.

Thus, Q' simulates each step of Q in $O(T(n))$ steps, and the entire computation of Q in $O(T^2(n))$ steps. □

We construct a PRAM[$*, \uparrow$], MS, that simulates Q via Q' in $O(\log T(n))$ time.

• *Construction of a description of all configuration sequences of* Q'

Let d denote the number of bits needed to encode each symbol in the tape alphabet. Let σ be a configuration of a one-tape NTM. Let η_1 be a bit string describing the contents of the worktape in σ with d bits per symbol from the right end to the square that the tape head is reading (but not including the contents of that square); let η_2 be a bit string describing the contents of the worktape from the square that the tape is reading to the left end of the tape; let ϕ denote an encoding of *state* (σ). We encode a configuration σ by an integer μ such that $\#\mu = \eta_2 \phi \eta_1$.

Since Q' runs for $O(T^2(n))$ time on $T(n) + 2$ space, each configuration of Q' can be described in $O(T(n))$ bits, and each computation of Q' can be described in $O(T^3(n))$ bits. Let I_m denote the integer such that $\#I_m$ is the concatenation of all bit strings m bits long. Specifically,

$$I_m = \sum_{i=0}^{2^m - 1} i \cdot 2^{im}.$$

For an integer x, where $\#x = b_y \cdots b_1 b_0$, for all $0 \le i \le \lfloor y/m \rfloor$, we call $b_{(i+1)m-1} \cdots b_{im}$, a *slot*. For $m = O(T^3(n))$, *MS* will generate I_m as a list of all strings that can possibly describe a computation of Q' on input ω. We view $\#I_m$ as the concatenation of 2^m slots, each of which represents a sequence of configurations of Q'.

We defined I_m as the sum of 2^m terms. We want *MS* to generate I_m in $O(\log m)$ time. Therefore, we cannot simply activate a processor to build each term, then sum the terms because this process takes $O(m)$ time.

Let $mask_j = \sum_{i=0}^{2^m - 1} 2^{im+j}$; that is, $\#mask_j$ has a 1 in the jth bit position of each slot. Let $S_j = I_m \wedge mask_j$; that is, $\#S_j$ and $\#I_m$ are equal in the jth bit position of each slot, and $\#S_j$ is

0 elsewhere. Recall that slot k of $\#I_m$ contains $\#k$, so the value of the jth position in slot k of $\#S_j$ is equal to the value of the jth bit position of $\#k$.

MS generates I_m by building each S_j, $0 \le j \le m-1$, then combining them: $I_m = \bigvee_{j=0}^{m-1} S_j$.

Let us now describe how *MS* constructs the S_j's. Although we have defined S_j in terms of I_m, we utilize an alternative definition to explain our construction of S_j. Let $b = 2^m/2^{j+1}$. Let $river_j = \sum_{i=0}^{b-1} 2^{(2i+1)m\,2^j + j} = \sum_{i=0}^{b-1} 2^{im\,2^{j+1} + m\,2^j + j}$ and $bayou_j = \sum_{k=0}^{2^j-1} 2^{km}$. Thus, $S_j = river_j \cdot bayou_j$. If we look only at the jth bit position of each slot of $\#I_m$ (or equivalently, $\#S_j$), we find alternating sequences of 2^j 0's and 2^j 1's. Our second definition of S_j reflects this: we place a 1 in the jth position of each (2^{j+1})th slot ($river_j$), then multiply by an integer whose two's complement representation has 2^j 1's, appropriately spaced ($bayou_j$).

To generate each $river_j$, *MS* combines a set of values called $stream_k$. We now define $stream_k$ and tell how *MS* generates the $stream_k$'s and how *MS* uses them to build the $river_j$'s. For all $0 \le k \le m-1$, $stream_k = 2^{m\,2^k} + 1 = [1 \uparrow (m \cdot 2^k)] + 1$; that is, $\#stream_k$ has a 1 in the rightmost position of slot 2^k and a 1 in the rightmost position of slot 0. *MS* activates m processors, P_m, ..., P_{2m-1}, in $O(\log m)$ steps. Processor P_{m+k}, $1 \le k \le m-1$, computes $stream_k$ in constant time.

Next, we generate $\tilde{}river_j$, a value one step from $river_j$. Define $\tilde{}river_j = \sum_{i=0}^{b-1} 2^{im\,2^{j+1}}$; that is, $\tilde{}river_j$ is $river_j$ shifted right so that the least significant 1 is in the 0th bit position: $\tilde{}river_j = river_j \downarrow (m\,2^j + j)$. We build $\tilde{}river_j$, $0 \le j \le m-2$, as the product of $m - j - 1$ $stream_k$'s: $\tilde{}river_j = \prod_{k=j+1}^{m-1} stream_k$. We want to compute all $\tilde{}river_j$'s in $O(\log m)$ time. This is a parallel postfix computation, which *MS* can perform in time $O(\log m)$ with m processors (Ladner and Fischer, 1980).

We now have $\tilde{}river_j$ for $0 \le j \le m-2$. The remaining item, $\tilde{}river_{m-1}$, is simply 1. Now for all $0 \le j \le m-1$, we obtain $river_j$ from $\tilde{}river_j$ in constant time: $river_j \leftarrow \tilde{}river_j \uparrow (m2^j + j)$.

MS next computes the $bayou_j$'s. Each $bayou_j$ is the product of j easily computed terms:

$$bayou_j = \sum_{k=0}^{2^j-1} 2^{km} = \prod_{s=1}^{j} 2^{(m/2)2^s} + 1.$$

This is a parallel prefix computation. Once again, *MS* computes all $bayou_j$'s simultaneously in $O(\log m)$ time (Ladner and Fischer, 1980).

For all j, $0 \le j \le m-1$, processor P_{m+j} computes $S_j = river_j * bayou_j$. *MS* now computes $I_m = \bigvee_{j=0}^{m-1} S_j$ in $O(\log m)$ time.

Since $m = O(T^3(n))$, *MS* computes $I_{T^3(n)}$ in $O(\log T(n))$ time.

• *Testing for a valid computation*

We view $I_{T^3(n)}$ as a list of all possible *configuration sequences*. Each sequence is $O(T^2(n))$ configurations long, and each configuration is $O(T(n))$ bits long. Thus, $I_{T^3(n)}$ is a list of all possible descriptions of a computation of the oblivious sweeping NTM Q' on the input. *MS* must now test whether at least one of the configuration sequences in $\#I_{T^3(n)}$ represents an accepting computation of Q'. *MS* will first build a set of bit masks to be used in the testing, then for each configuration sequence, *MS* will evaluate the following:

Test 1: whether the transition from each configuration to the next is valid,

Test 2: whether the first configuration corresponds to the initial configuration of Q', and

Test 3: whether Q' is in an accepting state in the last configuration.

If all tests are true for a configuration sequence, then *MS* accepts; if for all configuration sequences, at least one test fails, then *MS* rejects.

Recall that Q' runs for $O(T^2(n))$ steps on $T(n) + 2$ space, so a computation comprises $p = O(T^2(n))$ configurations, where each configuration is $f = O(T(n))$ bits long. The tape head of Q' sweeps back and forth between the endmarkers: the tape head makes $T(n) + 1$ moves from one endmarker to the next, then reverses direction. Hence, for $0 \le i \le p/(2T(n)+2)$ and $0 \le j \le T(n)$, in configurations numbered $i(2T(n)+2) + j$, the head is located one square to the left in the following configuration, and in configurations numbered $i(2T(n)+2) + T(n) + 1 + j$, the head is located one square to the right in the following configuration.

We interpret each slot of $\#I_m$ as a representation of a configuration sequence of Q'. Further, we view each slot of $\#I_m$ as the concatenation of p *notches*, where each notch represents a configuration of Q'. Let *config* (j) denote the contents of notch j interpreted as a configuration. Let us call the notches that hold configurations from which the tape head is to move right (left) *right notches* (*left notches*). For $0 \le i \le p/(2T(n)+2)$ and $0 \le k \le T(n)$, left notches are numbered $i(2T(n)+2) + k$; right notches are numbered $i(2T(n)+2) + T(n) + 1 + k$. (Right and left notches occur in alternating sequences of $T(n) + 1$ notches.) A right (left) notch describes one configuration in a configuration sequence during a right (left) sweep of Q'.

Recall that $N_l(\sigma)$ ($N_r(\sigma)$) denotes the neighborhood in configuration σ in which the tape head is on the left (right) square. Given the contents of two adjacent notches that

represent configurations σ_i and σ_{i+1}, where the tape head of Q' is in square q in configuration σ_i, MS must check the following to determine whether $\sigma_i \vdash \sigma_{i+1}$: (1) MS must compare $N_l(\sigma_i)$ and $N_r(\sigma_{i+1})$ (if $state(\sigma_i) \in R$) or $N_r(\sigma_i)$ and $N_l(\sigma_{i+1})$ (if $state(\sigma_i) \in L$) to determine whether the neighborhoods around the tape head in σ_i and σ_{i+1} represent a valid transition of Q' and (2) MS must check that the remainder of the configuration is unchanged. To perform Test 1, MS builds four values to be used as bit masks: $Lmask_0$, $Lmask_1$, $Rmask_0$, and $Rmask_1$. For the number j of a left (respectively, right) notch, $\#Lmask_0$ (respectively, $\#Rmask_1$) will have 1's in the bit positions corresponding to $N_r(config(j))$ if j is even and $N_l(config(j))$ if j is odd, and the other notches will be all 0's. For the number j of a left (respectively, right) notch, $\#Lmask_1$ (respectively, $\#Rmask_0$) will have 1's in the bit positions corresponding to $N_l(config(j))$ if j is even and $N_r(config(j))$ if j is odd, and the other notches will be all 0's. Thus, $Lmask_0$ and $Lmask_1$ test transitions from the left notches (transitions in a left sweep), and $Rmask_0$ and $Rmask_1$ test transitions from the right notches (transitions in a right sweep).

In Figure 7.1, the squares represent a set of four adjacent left notches, the arrows indicate the squares that the tape head should be reading in each configuration, the x's represent 1's in $\#Lmask_0$, and the y's represent 1's in $\#Lmask_1$. We use $Lmask_0$ to test transitions from even numbered left notches. $Lmask_0$ isolates a constant size neighborhood around the tape head to test the transition, and $\overline{Lmask_0}$ isolates the remainder of the configuration to test that it remains unchanged.

We describe the construction of $Lmask_0$; MS constructs the other masks similarly. Let us first formally define $Lmask_0$ over a single slot. A slot comprises p notches, each f bits long. In left notches j and $j+1$, for j even, $\#Lmask_0$ has 1's covering identical positions, then in left notches $j+2$ and $j+3$, $\#Lmask_0$ has 1's covering positions two squares to the left. In notch j, the tape head is on the right of the two squares in the covered neighborhood; in

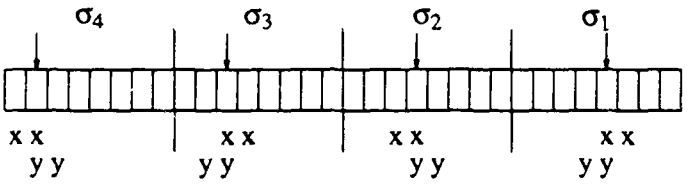


Figure 7.1. Portions of $Lmask_0$ and $Lmask_1$

notch $j+1$, the tape head is on the left of the two squares in the covered neighborhood. (That is, 1's cover the bit positions corresponding to $N_l(config(j))$ and $N_r(config(j+1))$.)

Assume without loss of generality that $T(n)$ is even. Let d denote the number of bits needed to specify each symbol in the tape alphabet, and let e denote the number of bits needed to specify a neighborhood.

We will build $Lmask_0$ from $mask_c$ (defined below):

$$Lmask_0 = mask_c \vee (mask_c \uparrow f)$$

since $\#Lmask_0$ has 1's in identical positions in adjacent pairs of left notches. We build our definition of $mask_c$ from a definition over a set of left notches, then a definition over a single slot, then a full-size definition. Let us define the following masks. Recall that $m = O(T^3(n))$.

$$mask_a = \sum_{k=0}^{T(n)/2} 2^{k(2f+2d)} + 2^{k(2f+2d)+1} + \cdots + 2^{k(2f+2d)+e-1}$$

$$mask_b = \sum_{j=0}^{p/(2T(n)+2)} (2^{jd(2T(n)+2)} \cdot mask_a) = (\sum_{j=0}^{p/(2T(n)+2)} 2^{jd(2T(n)+2)}) \cdot mask_a$$

$$mask_c = \sum_{i=0}^{2^m-1} (2^{ip} \cdot mask_b) = (\sum_{i=0}^{2^m-1} 2^{ip}) \cdot mask_b$$

Note that $mask_a$ is the portion of $mask_c$ over only a single set of left notches, and $mask_b$ is the portion of $mask_c$ over only a single slot.

Now let us describe how MS constructs $mask_c$. First, MS constructs $mask_a$ in $O(\log T(n))$ time, using $fT(n)/2$ processors. MS next computes $\sum_{j=0}^{p/(2T(n)+2)} 2^{jd(2T(n)+2)}$. Observe that $p/(2T(n)+2) = O(T(n))$, so MS also computes this quantity in $O(\log T(n))$ time. MS now obtains $mask_b$ with a single multiplication. Finally, MS builds $\Xi = \sum_{i=0}^{2^m-1} 2^{ip}$. This is more difficult, since Ξ is the sum of 2^m terms, and MS must build Ξ in $O(\log T(n)) = O(\text{loglog } 2^m)$ time. We rely on an interaction between multiplication and shift, as in our construction of I_m. We will show that Ξ is also the product of m easily computed terms. Define $brook_k = 2^{(p/2)2^k} + 1$. Then

$$\Xi = \sum_{i=0}^{2^m-1} 2^{ip} = \prod_{k=1}^{m} brook_k .$$

MS computes $brook_k$, $1 \le k \le m$, in constant time, then builds Ξ in $O(\log m)$ time with m processors. Now we build $mask_c = \Xi \cdot mask_b$ in one step. MS next builds $Lmask_0 = mask_c \vee (mask_c \uparrow f)$. MS constructs $Lmask_1$, $Rmask_0$, and $Rmask_1$ similarly in $O(\log T(n))$ time.

We now describe how we use $Lmask_0$ to test whether half the transitions from configurations in left notches are valid. (We will use $Lmask_1$ to test the other half.) For each j, where j is even and the number of a left notch, we test whether $config(j) \vdash config(j+1)$. Simulator MS performs the tests for all such configurations simultaneously.

Since a TM has a finite-state control and a neighborhood can be specified in a constant number of bits, there are only a constant number of valid transitions. Let N denote the set of all e bit strings γ such that γ contains a description of two tape symbols written by Q' and a description of a state of Q' at the position of the tape head. That is, N is the set of strings

describing neighborhoods that can actually occur in a computation of Q'. Let TR denote the set of pairs (ζ, η) such that $\zeta, \eta \in N$, and in one step of Q', neighborhood ζ with Q' in the state and with cell contents specified by ζ, becomes neighborhood η; that is, TR is the set of valid transitions between neighborhoods. For each $(\zeta, \eta) \in TR$, MS creates masks $\zeta mask = \zeta \cdot \Xi$ and $\eta mask = \eta \cdot (\Xi \uparrow f)$.

Let j be the number of a left notch and let j be even. Let $yang(j)$ denote the portion of notch j where the tape head of Q' should be in a computation (the position of $N_l(config(j))$); let $yin(j)$ denote the rest of notch j. For each pair $(\zeta, \eta) \in TR$, MS tests $yang(j)$ for equality with ζ, for all such j, by using $\zeta mask$. For those found equal, MS tests $yang(j+1)$ for equality with η using $\eta mask$, and MS tests $yin(j)$ and $yin(j+1)$ for equality using $\overline{Lmask_0}$. For those found equal, MS tags notch $j+1$ with a 1. With a constant number of such tests, MS checks the transitions from even numbered left notches j.

Similarly, MS tests the remainder of the transitions from all notches in all slots of I_m using $Lmask_1$, $Rmask_0$, and $Rmask_1$. This completes Test 1.

MS now performs Test 2 and Test 3. Recall that Test 2 is a test of whether the first notch in each slot contains the initial configuration of Q', and Test 3 is a test of whether the last notch in each slot contains an accepting configuration of Q'. In $O(\log T(n))$ time, MS builds masks for each of these tests in the same manner as its other masks, then performs Tests 2 and 3 in constant time, tagging those notches that pass the test with a 1 and those notches that fail with a 0.

For each slot, MS now ANDs all its tags together (the tags written during the tests). All the tags for all the slots are concatenated in a single cell $c(g)$. If any notch in a slot fails one of the tests, then the AND of tags in that slot is 0. If every notch in a slot passes every test,

then the AND of tags in that slot is 1. If $con(g) = 0$, then some notch in every slot has failed a test, and no slot holds a valid computation; therefore, *MS* rejects the input. If $con(g) \neq 0$, then some slot holds a representation of a valid computation; therefore, *MS* accepts the input.

Theorem 7.1. For all $T(n) \geq \log n$, $NTIME(T(n)) \subseteq PRAM[*,\uparrow]\text{-}TIME(\log T(n))$.

Proof. By the simulation above, a PRAM[*,↑], *MS*, simulates an NTM, Q, via an oblivious sweeping NTM, Q'. Q' simulates Q in time $O(T^2(n))$, and *MS* simulates Q', hence Q, in time $O(\log T(n))$. □

Corollary 7.1.1. $NEXPTIME \subseteq PRAM[*,\uparrow]\text{-}PTIME$.

7.2. Simulation of PRAM[*,↑,↓] by TM

By itself, the shift operation produces numbers that can be extremely long, but not very complex. We took advantage of this lack of complexity by manipulating encodings of numbers when we simulated a PRAM[↑,↓] in Chapter 6. By itself, the multiplication operation produces numbers that are more complex, but reasonably bounded in length. We took advantage of this bound on length by addressing individual bits of numbers generated with multiplication when we simulated a PRAM[*] in Chapter 4. The combination of multiplication and shift gives rise to extremely long numbers of greater complexity. To simulate a PRAM[*,↑,↓], a TM once again uses the interesting bit encoding to deal with the length; the increased complexity of the numbers requires the TM to use more space.

If $i = j*k$, where j has x_j interesting bits, and k has x_k interesting bits, then i has up to $x_j \cdot x_k$ interesting bits. Consequently, an encoding of a number generated by a PRAM[*,↑,↓] may have a doubly exponential number of nodes. Simon (1981a) gave only a short sketch as

a proof that $RAM[*,\uparrow]$–$PTIME \subseteq EXPSPACE$. He gave the bare bones of a proof of a containment in doubly exponential space by writing out entire encodings, then said that the space bound could be reduced to singly exponential space by using pointers into an encoding. Here we present the details of a simulation of a PRAM[*,↑,↓] running in polynomial time by a Turing machine running in exponential space. This simulation is similar to the direct simulation of a PRAM[↑,↓] running in polynomial time on a Turing machine running in polynomial space (Section 6.3) in that we manipulate pointers into the interesting bit encoding of cell contents. Since the number of interesting bits is at most doubly exponential, the TM uses exponential space to describe a pointer into an encoding.

Let Q be a PRAM[*,↑,↓] that uses $T(n)$ time and $P(n)$ processors. Let Q' be a PRAM[*,↑,↓] that uses only short addresses and simulates Q according to the Associative Memory Lemma. Thus, Q' uses $O(P^2(n)T(n))$ processors, $O(T(n))$ time, and only addresses in 0, 1, ..., $O(P(n)T(n))$.

We construct a TM M that simulates Q via Q' in $O(T^2(n)4^{T(n)} \log n)$ space. Without loss of generality, assume M is given $T(n)$.

M uses five mutually recursive procedures: *PCOUNTER*, *COMPARE*, *SYMBOL*, *ADD*, and *MULTIPLY*. The first four of these are the procedures of the same names from Section 6.3, the direct simulation of a PRAM[↑,↓] by a TM, except that if *instr* is $r(i) \leftarrow r(j) * r(k)$ in *SYMBOL*, then $SYMBOL(j, \psi, m, t)$ returns $MULTIPLY(j, k, \psi, m, t-1)$. Every parameter can be written in the space required to write a pointer into an encoding. (This space will be bounded in Lemma 7.2.3.)

When we add a column of partial products while performing a multiplication, the column can include up to z 1's in it, where

$$z = O\left(2^{2^{\cdots^{2}}}\right) T(n)\uparrow\downarrow -1$$

is the operand length. To express the sum of the column, we would have to represent every integer from 0 to z, and we cannot represent all such numbers in exponential space, regardless of the representation. We use the Booth multiplier encoding algorithm (Hwang, 1979) to overcome this obstacle.

Let d be an integer and let $w = len(d)$. Let $b_{w-1} \cdots b_0$ be the w-bit two's complement representation of d. Define $b_{-1} = 0$. A *plus Booth bit* of d is a bit b_i such that $b_i = 0$ and $b_{i-1} = 1$; a *minus Booth bit* of d is a bit b_i such that $b_i = 1$ and $b_{i-1} = 0$. A *Booth bit* of d is a bit that is either a plus Booth bit or a minus Booth bit.

We define the *Booth (B) encoding* as $B(d) = c_{w-1} \cdots c_0$, where $c_i = 1$ if b_i is a plus Booth bit, $c_i = \bar{1}$ (-1) if b_i is a minus Booth bit, or $c_i = 0$ otherwise. The Booth encoding replaces strings of 1's with a 1, 0's, and a $\bar{1}$ (-1).

For example, $B(01111111) = 1000000\bar{1}$. Suppose we want to multiply a multiplicand m by multiplier 01111111. By the naive algorithm, we add seven partial products, each of which is m shifted by some value; by the Booth algorithm, we add only $m \uparrow 7$ and $-m$.

We define the *Booth-interesting (BI) encoding*, by

$$BI(0) = 0,$$

$$BI(1) = 1,$$

$$BI(d) = (I(a_k), ..., I(a_1); r),$$

where a_j is the position of the jth Booth bit of d and r is the value (0 or $\bar{1}$) of the rightmost digit of $B(d)$.

The Booth-interesting encoding and the interesting bit encoding are closely related: an interesting bit at position j of $\#d$ corresponds to a 1 or $\bar{1}$ in $B(d)$ at position $j+1$. The

rightmost nonzero in B(d) is a $\bar{1}$, with 1 and $\bar{1}$ alternating afterwards. If #d begins with a 1 in the rightmost position, then B(d) has a $\bar{1}$ in the rightmost position; otherwise, B(d) has a 0 in the rightmost position. Because of this relationship, M readily converts from I(d) to BI(d) with procedure *ADD*.

To perform a multiplication, we will convert the multiplier from the interesting bit encoding to the Booth-interesting encoding, then multiply it with the multiplicand. By Booth's algorithm, we have only as many partial products as we have Booth bits. Since each integer in a computation has at most $O(n^{2T(n)})$ interesting bits (Lemma 7.2.3), there will be $O(n^{2T(n)})$ partial products.

We now describe *MULTIPLY*(j, k, ψ, m, t), where j and k are register addresses, ψ is a pointer, m is a processor number, and t is a time step. *MULTIPLY*(j, k, ψ, m, t) returns the symbol to which ψ points in I($val(rcon_m(j))$*$val(rcon_m(k))$) if ψ points to a leaf in the product; otherwise, *MULTIPLY* returns a signal that ψ points to a subtree of the product. Assume that we are manipulating pointers into the Booth-interesting encoding of the multiplier, $val(rcon_m(k))$. Let γ denote I($val(rcon_m(j))$*$val(rcon_m(k))$).

Using the Booth encoding of the multiplier, we have $O(n^{2T(n)})$ partial products. Each partial product ρ is a copy of the multiplicand shifted by the value of the position of a Booth bit, b, in the multiplier. If b is a plus Booth bit, then ρ is added; if b is a minus Booth bit, then ρ is subtracted. To efficiently find the symbol $\gamma.\psi$, we perform carry-save addition of the partial products. This simplifies our computations, since if $t \leftarrow u \bigcirc v$, where $\bigcirc$ is a Boolean operation, then each first-level subtree of I(t) is a first-level subtree of I(u) or I(v).

To perform carry-save addition on three numbers, t, u, and v, we generate a sum term S and a carry term C by Boolean operations on t, u, and v. Then we add S and C to get

$t + u + v$. Specifically, $S = t \oplus u \oplus v$ and $C = (majority\,(t,\ u,\ v)) \uparrow 1$, where $(majority\,(t,\ u,\ v))$ returns 0 (1) in position y if the majority of bits in position y of $\#t$, $\#u$, and $\#v$ are 0 (1).

We wish to compute the sum of $g = n^{2^{T(n)}}$ operands, $OP_1, OP_2, ..., OP_g$. We use a divide-and-conquer method, repeatedly splitting each sum of y operands into two sums of $y/2$ operands, until $y = 2$. When we reach the base case, we declare one operand to be S and the other to be C, then return one level up in the recursion. At this level, two partial sums return their sum and carry terms: S_1 and C_1 and S_2 and C_2. We produce sum and carry terms, S_3 and C_3, for $S_1 + C_1 + S_2$, then sum and carry terms, S_4 and C_4, for $S_3 + C_3 + C_2$. Then S_4 and C_4 are returned up to the next level of recursion. At the end of the recursion, we have a sum term S_g and a carry term C_g, both of which were generated solely by Boolean operations, and we add these together.

This description overlooks one important factor: just as we do not have enough space to write an integer generated by a PRAM[*,↑,↓] or even its encoding, we do not have enough space to write a sum or carry term. The key here is keeping track of subtrees through the recursion. Instead of viewing this problem from the bottom as combining sum and carry terms until we obtain S_g and C_g, let us view the problem from the top. M wants to obtain the symbol indicated by the pointer ψ into the encoding of the sum of the partial products γ and is using carry-save; consequently, $val\,(\gamma) = S_g + C_g$. By our rules of addition (Appendix E), M needs symbols in S_g and in C_g. Suppose M wants a particular symbol in S_g. By the procedure described above, S_g is the XOR of three sum and carry terms; therefore, M looks for symbols in each of the terms contributing to S_g according to our rules for XOR. (These rules are similar to the rules for OR in Appendix D.) M continues in this manner until it

reaches a base case, which is handled by our previously described rules. The recursion stops after $O(\log g) = O(n2^{T(n)})$ levels. This completes the description of *MULTIPLY*.

We now present lemmas, analogous to those of Section 6.1, that bound the length of a pointer into an encoding. Lemma 7.2.1 bounds the depth of an encoding and the number of interesting bits in a number generated by a PRAM[*,↑,↓]. Let *bool* be a set of Boolean operations.

Lemma 7.2.1. If a processor P_m executes $r(i) \leftarrow r(j) * r(k)$, then $depth(rcon_m(i)) \leq 2 + \max\{depth(rcon_m(j)), depth(rcon_m(k))\}$ and $intbits(rcon_m(i)) \leq intbits(rcon_m(j)) * (1 + intbits(rcon_m(k)))$.

Proof. The product $rcon_m(i)$ is the sum of partial products. Each nonzero partial product is $rcon_m(j)$ shifted left by the value of the position of a 1 bit in $rcon_m(k)$. We can add the partial products by carry-save addition. Thus, we can perform a series of Boolean operations on the partial products and a single addition at the end. By Parts i) and iv) of Lemma 6.1.1, $depth(rcon_m(i)) \leq 2 + \max\{depth(rcon_m(j)), depth(rcon_m(k))\}$. Recall that we convert the multiplier to the Booth-interesting encoding in the procedure *MULTIPLY*. For an integer d, the number of Booth bits in $B(d)$ is at most $1 + intbits(d)$; therefore, Parts i) and ii) of Lemma 6.1.1 apply to the number of Booth bits in $B(d)$. Thus, the number of Booth bits in $B(rcon_m(k))$ is $1 + intbits(rcon_m(k))$. Therefore, $rcon_m(i)$ is the sum of $1 + intbits(rcon_m(k))$ nonzero partial products, each with $intbits(rcon_m(j))$ interesting bits. Therefore by Part i) of Lemma 6.1.1, $intbits(rcon_m(i)) \leq intbits(rcon_m(j)) * (1 + intbits(rcon_m(k)))$. □

Part i) of Lemma 7.2.2 bounds the number of subtrees below first level nodes in an encoding; Part ii) bounds the number of subtrees below fth level nodes in an encoding, $f > 1$.

Lemma 7.2.2. Suppose a processor P_m executes $r(i) \leftarrow r(j)\ O\ r(k)$, where $O \in \{+, \uparrow, \downarrow, *, -, bool\}$, $I(rcon_m(i)) = (I(a_r), ..., I(a_1); w_i)$, $I(rcon_m(j)) = (I(b_s), ..., I(b_1); w_j)$, and $I(rcon_m(k)) = (I(c_t), ..., I(c_1); w_k)$, where a_v, b_v, c_v denote the positions of the vth interesting bits of $rcon_m(i)$, $rcon_m(j)$, and $rcon_m(k)$, respectively.

i) For $I(a_v)$ (that is, the vth subtree at level 1 of $I(rcon_m(i))$), if O is *, then $intbits(a_v) \leq \max_q \{intbits(b_q)\} + \max_q \{intbits(c_q)\}$.

ii) For $I(\beta)$ a subtree at level $f > 1$, $intbits(\beta) \leq 1 + \max_q \{intbits(q\text{th subtree of } rcon_m(j) \text{ at level } f), intbits(q\text{th subtree of } rcon_m(k) \text{ at level } f)\}$.

Proof. i) A first level subtree $I(a_v)$ encodes the position p of an interesting bit in $rcon_m(i)$. The subtrees of $I(a_v)$ encode the positions of interesting bits in p. Suppose P_m executes $r(i) \leftarrow r(j) * r(k)$. Recall that we convert the multiplier to the Booth-interesting encoding in procedure *MULTIPLY* and that for an integer d, the position of a Booth bit in $B(d)$ is exactly 1 beyond the position of an interesting bit in d. The product $rcon_m(i)$ is the sum of partial products of $rcon_m(j)$ and $B(rcon_m(k))$. Each nonzero partial product is either plus or minus $rcon_m(j)$ shifted left by the value of the position of a Booth bit in $B(rcon_m(k))$. By Part i)b) of Lemma 6.1.2 and the relationship between Booth bits and interesting bits, the number of Booth bits in the position of a 1 or $\bar{1}$ in a partial product is at most $\max_q \{intbits(b_q)\} + \max_q \{intbits(c_q)\}$. By Part i)a) of Lemma 6.1.2, the position of an interesting bit in $rcon_m(i)$ has the same upper bound.

ii) For any instruction, we add at most 1 to the value of a subtree of level $f > 1$, so Part i)a) of Lemma 6.1.2 applies. □

Lemma 7.2.3. A pointer used by M can be specified in $O(T(n)\, 2^{T(n)} \log n)$ space.

Proof. Let d be an integer generated by Q. By Lemmas 6.1.1 and 7.2.1, $depth(d) \leq 2T(n)$. If ω is the input to Q and ω has length n, then $intbits(\omega) \leq n$. Let $I(\beta)$ be $I(d)$ or a subtree of $I(d)$. By Lemmas 6.1.1, 6.1.2, 7.2.1, and 7.2.2, $intbits(\beta) \leq n^{2^{T(n)}}$. Therefore, any leaf in $I(d)$ can be specified by a pointer of length $T(n)\, 2^{T(n)} \log n$. (The tree has $T(n)$ levels, and we need space $2^{T(n)} \log n$ to specify the branch at each level.) □

Theorem 7.2. For all $T(n) \geq \log n$, $PRAM[*, \uparrow, \downarrow]\text{-}TIME(T(n)) \subseteq$ $DSPACE(T^2(n)\, 4^{T(n)}\, n \log n)$.

Proof. By the simulation given above, a TM M simulates a PRAM[*,↑,↓] Q via Q'.

By Lemma 7.2.3, for each invocation of *PCOUNTER*, *COMPARE*, *SYMBOL*, *ADD*, and *MULTIPLY*, M can write all variables and parameters in space $O(T(n)\, 2^{T(n)} \log n)$. The depth of recursion of the first four of these procedures is at most $O(T(n))$. The depth of recursion of *MULTIPLY* is at most $O(n2^{T(n)})$ for each invocation. Thus, M can simulate Q via Q' in space $O(T^2(n)\, 4^{T(n)}\, n \log n)$. □

Corollary 7.2.1. $PRAM[*, \uparrow, \downarrow]\text{-}PTIME \subseteq EXPSPACE$.

Chapter 8. Probabilistic Choice

In this chapter, for various instruction sets *op*, we present simulations of probabilistic PRAM[*op*]s by deterministic PRAM[*op*]s. We also relate probabilistic unbounded fan-in circuits and CRCW *prob*-PRAMs.

8.1. Background

Much attention and study have been devoted to probabilistic, or randomized, algorithms in the past several years. Survey papers by Rabin (1976), Welsh (1983), and Rajasekaran and Reif (1987) present a sampling of the work done on probabilistic algorithms. Naturally, the probabilistic Turing machine (PTM) is the foundation for many probabilistic algorithms. Gill (1977) defined the PTM as tossing a coin to decide state transitions. He also defined different restrictions on language recognition: 1-sided, bounded 2-sided, and 2-sided error. *PSPACE* on PTMs is equivalent to *PSPACE* on deterministic Turing machines (Simon, 1981b).

Reif (1984) presented simulations between *prob*-RAM[*,÷]s and *prob*-PRAM[*,÷]s. We define a configuration of a RAM to comprise the contents of each of the registers used in memory and the contents of the program counter of the processor.

Reif defined his model as follows. Let c be a constant. From any configuration C_i, a probabilistic machine Q may enter any configuration from the set $NEXT_i$ in one step, where $NEXT_i$ contains no more than c elements, c a constant. Q chooses each element of $NEXT_i$ with equal probability, independently of previous and succeeding choices. Q accepts an input string ω of length n in time $T(n)$ if the probability that a computation of Q on ω reaches an *ACCEPT* instruction within $T(n)$ steps is strictly greater than ½. We specify the

expression of the choice of next configuration as follows. Each instruction is distinctly numbered and has the form:

$$r(i) \leftarrow r(j) \bigcirc r(k); \rho_1, \rho_2, ..., \rho_f$$

in which $\rho_1, \rho_2, ..., \rho_f$ are integers denoting instruction labels, and $f \leq c$; the processor executes $r(i) \leftarrow r(j) \bigcirc r(k)$, then uniformly selects one of $\{\rho_1, \rho_2, ..., \rho_f\}$ as the next instruction. Thus a machine in a configuration such that it is currently executing the instruction above has a choice of f possible next configurations. We allow repetition of choices for next instruction (to weight the probability of selection).

Three theorems from Reif (1984) relating *prob*-RAM[*,÷]s and *prob*-PRAM[*,÷]s follow.

Theorem 8.1. (Reif, 1984) Let Q be a *prob*-RAM[*,÷] with constructible time bound $T(n) \geq n$, memory bound $S(n)$, and integer bound $I(n)$, where $S(n)$ bounds the number of registers used by Q and $I(n)$ bounds the value of the largest integer. Then there is a *prob*-PRAM[*,÷] PZ that simulates Q. If Q is unit-cost, then PZ has unit-cost time bound $O(S(n)\log I(n) + \log T(n))$ and processor bound $O(I(n)^{S(n)}T(n))$; if Q is log-cost, then PZ has log-cost time bound $O((S(n) + \log T(n))^2)$ and processor bound $O(4^{S(n)}T(n))$.

Proof (sketch). PZ activates one processor for each pair (χ, t), where χ is a configuration of Q and t is a time step. Let $NEXT_\chi$ denote the set of possible configurations reachable by Q from χ in one step. Each processor with pair (χ, t) randomly chooses some $\chi' \in NEXT_\chi$ and writes $(\chi', t+1)$. This gives the equivalent of a one-step transition matrix. PZ then computes the transitive closure of that matrix. □

Theorem 8.2. (Reif, 1984) Let PZ be a *prob*-PRAM[*,÷] with constructible time bound $T(n)$, memory bound $S(n)$, and processor bound $P(n)$. Then there is a *prob*-RAM[*,÷] Q

with memory bound $O(S(n) + P(n))$ simulating PZ. If PZ is unit-cost, then Q has unit-cost time bound $O(T(n)P(n))$; if PZ is log-cost, then Q has log-cost time bound $O(T(n)P(n)\log P(n))$.

Proof (sketch). Q performs a brute-force simulation, simulating one processor at a time. □

Theorem 8.3. (Reif, 1984) Let Q be a *prob*-RAM[*,÷] with constructible unit-cost time bound $T(n) \geq n$ and integer bound $I(n)$. There is a *prob*-PRAM[*,÷] PZ that simulates Q and has unit-cost time bound $O((T(n)\log T(n)\log(T(n)I(n)))^{1/2})$.

Proof (sketch). PZ partitions the $T(n)$ time steps into consecutive intervals of length L. The interval length is approximately $T^{1/2}(n)$. PZ then computes a look-up table that, for each configuration χ of Q, contains a configuration χ' reachable by Q from configuration χ in L steps. □

8.2. Choice Sequence Simulation

In this section, we simulate a *prob*-PRAM[*op*] by a deterministic PRAM[*op*]. The deterministic simulator evaluates all possible sequences of random choices in a computation of the *prob*-PRAM[*op*].

Let Γ_t be the set of processors of *prob*-PRAM PZ that makes a random choice at time t. We call the choice made by the ith lowest numbered processor in Γ_t the *ith random choice made by PZ at time t*. Suppose that the processors of PZ have made a total of j random choices before time t, and suppose that $k = i + j$. Then we call the choice made by the ith lowest numbered processor in Γ_t the *kth random choice made by PZ*. A computation of *prob*-PRAM PZ has *choice sequence* $a_0, a_1, ..., a_{r-1}$ if in the jth random choice, for all $0 \leq j \leq r-1$, a processor chooses the a_jth alternative.

In this section, we simulate a *prob*-PRAM[*op*] or *prob*-RAM[*op*] by a deterministic PRAM[*op*] D. Machine D will generate all possible choice sequences of the simulated machine, then determine the number of choice sequences that lead to acceptance.

For an integer x, let $\#x$ denote its two's complement representation. If $\#x = b_{r-1} \cdots b_1 b_0$, then let $<x>$ denote the sequence $b_0, b_1, ..., b_{r-1}$.

• *Sequential case*

For clarity of exposition, we present Theorem 8.4, the simulation of a sequential *prob*-RAM[*op*] by a deterministic PRAM[*op*]. The results stated in Theorem 8.4 are a corollary of Theorems 8.5 and 8.6 with $P(n) = 1$.

Theorem 8.4. Let $op \in \{\varnothing, \{*\}, \{*,+\}, \{\uparrow,\downarrow\}, \{*,\uparrow,\downarrow\}\}$. Let Q be a *prob*-RAM[*op*] with time bound $T(n)$ that makes $R(n)$ random choices. There is a deterministic PRAM[*op*] D that simulates Q in $O(T(n))$ time with $2^{R(n)}$ processors.

Proof. Suppose that each random choice made by Q is made between two alternatives. (At the end of the proof, we specify changes to the proof for a Q that is allowed more than two choices.) We construct a deterministic PRAM[*op*] D that simulates Q. Simulator D activates $2^{R(n)}$ processors in $O(R(n))$ time. These processors are numbered $2^{R(n)}$, ..., $2 \cdot 2^{R(n)} - 1$. Each processor number encodes a unique choice sequence of $R(n)$ elements. P_m computes $\sigma_m = m - 2^{R(n)}$. P_m will simulate a computation of Q with choice sequence $<\sigma_m>$.

P_m sets *mask* = 1. P_m will use *mask* to read bits of $\#\sigma_m$.

In simulating a general step of Q in which Q executes instruction *instr*, P_m does the following. If Q makes no random choices in *instr*, then P_m simply executes *instr*. If Q makes a random choice in *instr*, then P_m executes the portion of *instr* before making the

random choice, then uses *mask* to read the bit of σ_m indicating the outcome of the random choice. Next, P_m updates *mask* to prepare for the next random choice by adding *mask* to itself. Since *#mask* had a single 1 in the *j*th bit position, after the addition *#mask* has a single 1 in the $j+1$st bit position, enabling P_m to read the $j+1$st bit of σ_m when the next random choice arises.

In this manner, P_m simulates each step of Q with choice sequence $<\sigma_m>$ in constant time.

After simulating $T(n)$ steps of Q, P_m decides whether Q halts on an *ACCEPT* instruction. D now computes the number of computations that have accepted and the number that have not accepted, then uses this information to decide, based on the acceptance conditions, whether to accept or reject the input. This takes $O(R(n))$ time. The entire computation takes $O(R(n) + T(n))$ steps. Since $R(n) \leq T(n)$, D simulates Q in $O(T(n))$ time with $2^{R(n)}$ processors.

Now suppose Q can randomly select from up to c labels for the next instruction. Let d be the least common multiple of 1, 2, ..., c. Alter the instructions with random choices so that each has a choice of d labels. Let $e = \lceil \log d \rceil$. For an integer x, where $\#x = b_{r-1} \cdots b_1 b_0$, let $<x>'$ denote the sequence $a_0, a_1, ..., a_{r/e-1}$, such that $a_0 = b_e \cdots b_0$, $a_1 = b_{2e} \cdots b_{e+1}$, ..., $a_{r/e-1} = b_{r-1} \cdots b_{r-e}$. We say that a computation of Q has choice sequence $<x>'$ if, for all $0 \leq j \leq r/e - 1$, in the *j*th random choice made by Q from among d alternatives, Q chooses the a_jth alternative.

Let f be the least power of 2 greater than or equal to d. The deterministic PRAM[*op*] D activates $f^{R(n)}$ processors in $O(R(n))$ time. The processors are numbered $f^{R(n)}$, ..., $2 \cdot f^{R(n)} - 1$. Each processor number encodes a unique choice sequence of $R(n)$ elements.

P_m computes $\sigma_m = m - f^{R(n)}$. P_m will simulate a computation of Q with choice sequence $<\sigma_m>'$. In the course of simulating the computation as described above, if P_m reads an element a_i of $<\sigma_m>'$ such that $a_i > d$, then the choice sequence is invalid. P_m then does not report its computation as either accepting or rejecting. Of the $f^{R(n)}$ sequences, $d^{R(n)}$ are valid, exactly the number of choice sequences of Q with d choices per random instruction. □

• *Parallel case*

We extend the simulation of a sequential machine to a simulation of a parallel machine.

Theorem 8.5. Let $op \in \{\{*\}, \{*,\div\}, \{\uparrow,\downarrow\}, \{*,\uparrow,\downarrow\}\}$. Let PZ be a *prob*-PRAM[op] with time bound $T(n)$ and processor bound $P(n)$ that makes $R(n)$ random choices. There is a deterministic PRAM[op] D that simulates PZ in time $O(R(n) + T(n) \log P(n))$ with $P(n)2^{R(n)}$ processors.

Proof. Assume that each random choice made by PZ is made between two alternatives. If PZ makes random choices from among more than two alternatives, then modify the proof as described in the proof of Theorem 8.4.

We construct a deterministic PRAM[op] D that simulates PZ. D activates $2^{R(n)}$ processors in $O(R(n))$ time. These processors are numbered $2^{R(n)}, ..., 2 \cdot 2^{R(n)} - 1$. Each processor number encodes a unique choice sequence of $R(n)$ elements. Processor P_m computes $\sigma_m = m - 2^{R(n)}$. P_m will simulate a computation of PZ with choice sequence $<\sigma_m>$.

To simulate an access to shared memory cell $c(k)$ in a computation of PZ with choice sequence $<\sigma_m>$, D accesses its memory cell at address $k \cdot 2^{R(n)} + \sigma_m$. Given k, D can compute $k \cdot 2^{R(n)} + \sigma_m$ in constant time.

Assume without loss of generality that each processor P_m of D has two local memories: $lmem_1$ and $lmem_2$. P_m uses $lmem_1$ to simulate the memory of a processor of PZ and $lmem_2$ to perform its own computations.

Processor P_m, $2^{R(n)} \leq m \leq 2 \cdot 2^{R(n)} - 1$, simulates PZ with choice sequence $<\sigma_m>$. Processor P_m of D corresponds to P_0 of PZ until P_0 executes a *FORK* instruction. At this time, P_m executes a *FORK* instruction, halting and activating P_{2m} and P_{2m+1}. After this time, P_{2m} of D corresponds to P_0 of PZ and does not *FORK* any more, and P_{2m+1} corresponds to P_1 of PZ.

In simulating a general step of PZ in which processor P_k executes instruction *instr*, the corresponding processor P_j of D does the following. If P_k makes no random choices in *instr*, then P_j simply executes the instruction. If P_k makes a random choice in *instr*, then P_j must choose a bit of σ_m to decide the outcome of the random choice of PZ. Suppose W processors want to make a random choice at this step. The corresponding W processors of D sort themselves by processor number in $O(\log W)$ time (Cole, 1986). Suppose P_j is the vth lowest numbered processor wishing to make a random choice at this step. Then P_j reads the vth bit of $\#\sigma_m$. P_j uses this bit to decide whether to select the first or second label listed in *instr*. Processor P_{2m} then shifts σ_m right by W bits, leaving only the unread bits of σ_m. Note that $W \leq P(n)$.

In this manner, D simulates each step of PZ with choice sequence $<\sigma_m>$ in $O(\log P(n))$ time.

After simulating $T(n)$ steps of PZ, P_{2m} decides whether PZ halts on an *ACCEPT* instruction. D now computes the number of computations that have accepted and the number that have not accepted, then uses this information to decide, based on the acceptance

conditions, whether to accept or reject the input. This takes $O(R(n))$ time. The entire computation takes $O(R(n) + T(n) \log P(n))$ steps. □

Next, we simulate a *prob*-PRAM by a deterministic PRAM with the basic instruction set. The simulation is similar to the preceding one, except we do not interleave memory locations allocated to different choice sequences, and the basic PRAM must precompute tables of addresses and masks in order to access memory cells and the choice sequence.

Theorem 8.6. Let PZ be a *prob*-PRAM with time bound $T(n)$ and processor bound $P(n)$ that makes $R(n)$ random choices. There is a deterministic PRAM D that simulates PZ in time $O(R(n) + T(n) \log P(n))$ with $P(n)2^{R(n)}$ processors.

Proof. Given a *prob*-PRAM PZ, we construct a deterministic PRAM D that simulates PZ in $O(R(n) + T(n) \log P(n))$ time. The simulation follows that presented in the proof of Theorem 8.5 with three exceptions: (1) allocation of interleaved memory locations to different choice sequences, (2) addressing of memory cells, and (3) extraction of bits of $\#\sigma$.

Exception 1: Basic PRAM D cannot quickly compute $k \cdot 2^{R(n)} + \sigma_m$ for each memory access to cell k. Instead, D allocates a block of cells to each choice sequence. The processor assigned to each choice sequence initially computes the starting address of its block of cells. PZ can only access cells with addresses up to $O(n2^{T(n)})$. Thus, a block of cells of size $O(n2^{T(n)})$ is allocated to each choice sequence.

Exception 2: The addresses of the blocks assigned to each choice sequence range up to $O(n\,R(n)\,2^{T(n)})$. D computes a table of the addresses of the first cell of each block in $O(T(n) + \log R(n))$ time. D then uses this table to access the necessary cells in constant time.

Exception 3: D computes a set of $R(n)$ masks prior to beginning its simulation of PZ. For $0 \leq i \leq R(n) - 1$, $mask_i = 2^i$. D computes these masks in $O(R(n))$ time. In the previous simulation, processor P_{2m} shifted σ_m right by W bits after W processors read bits of σ_m. Here, D cannot perform right shifts, so P_{2m} keeps track of the last bit read of σ_m. With this information, a processor P_j wishing to make a random choice can select the appropriate bit of σ_m.

D spends $O(R(n) + T(n))$ initialization time and $O(\log P(n))$ time to simulate each step of PZ; hence, D simulates PZ in time $O(R(n) + T(n) \log P(n))$. □

8.3. Markov Chain Simulation

In this section, we present a simulation of a *prob*-PRAM[*,÷] PZ by a *deterministic* PRAM[*,÷] D in time $O((P(n) + \log I(n))\cdot S(n)\cdot \log(T(n)))$ with $O((k^{P(n)}I(n))^{3S(n)})$ processors. We achieve this by treating the computation of PZ as a finite Markov chain in which each configuration of PZ is a state. Depending on the relative values of $T(n)$, $S(n)$, and $P(n)$, this simulation may be more efficient for a PRAM[*,÷] than the simulations in the previous section.

Lemma 8.7.1. (Associative Memory Lemma for *prob*-PRAMs) Let $op \subseteq \{*, \div, \uparrow, \downarrow\}$. For all $T(n)$ and $S(n)$, every language recognized in time $T(n)$ using at most $S(n)$ cells by a *prob*-PRAM[*op*] PZ can be recognized in time $O(T(n))$ by a *prob*-PRAM[*op*] PZ' that uses $O(P(n)S(n))$ processors and accesses only cells with addresses in $0, \ldots, O(S(n))$.

Proof. The proof follows along the same lines as the proof of the Associative Memory Lemma (Chapter 3), with an extension to account for probabilistic choice. Replace $P(n)T(n)$ with $S(n)$, since in the proof of the Associative Memory Lemma $P(n)T(n)$ is used as a bound on the number of accessed cells.

When a processor P_g of PZ executes an instruction $r(i) \leftarrow r(j)\ \bigcirc\ r(k); \rho_1, \cdots, \rho_f$, it reads $rcon_g(j)$ and $rcon_g(k)$, computes $v := rcon_g(j)\ \bigcirc\ rcon_g(k)$, writes v in $r_g(i)$, and uniformly selects one of $\{\rho_1, \cdots, \rho_f\}$ as the next instruction. The corresponding processor P_m of PZ' simulates the first three parts of this step just as described in Chapter 3. P_m performs the fourth part by uniformly selecting one of $\{\rho_1, \cdots, \rho_f\}$. The time bound follows directly. □

Theorem 8.7. Let PZ be a *prob*-PRAM[*,÷] with time bound $T(n)$, processor bound $P(n)$, memory bound $S(n)$, integer bound $I(n)$, and program length k. Then there is a deterministic PRAM[*,÷] D that simulates PZ in time $O((P(n) + \log I(n))\cdot S(n)\cdot \log(T(n)))$ with $O((k^{P(n)}I(n))^{3S(n)})$ processors.

Proof. Let PZ' simulate PZ according to Lemma 8.7.1. Then PZ' has time bound $O(T(n))$, processor bound $O(P(n)S(n))$, memory bound $S(n)$, and integer bound $I(n)$.

Fix an input of length n. PZ' has $O((k^{P(n)}I(n))^{S(n)})$ distinct configurations with memory bound $S(n)$, since the value of the program counter of every processor must be considered; each can be encoded by a distinct integer no more than $O((k^{P(n)}I(n))^{S(n)})$.

Each instruction in the program of PZ' has up to c choices of next instruction. Let d be the least common multiple of the numbers of choices. D will view each instruction of PZ' as having d choices by duplicating all choices. This view preserves the probability of selecting each of the original choices.

D activates $(k^{P(n)}I(n))^{2S(n)}$ processors in $O(S(n)(P(n) + \log I(n)))$ time, one processor for each pair of configurations. The processor number of each of the $(k^{P(n)}I(n))^{2S(n)}$ processors encodes a pair (τ, υ), where τ and υ denote configurations of PZ'. D builds a transition probability matrix A. The processor associated with pair (τ, υ)

counts the number of ways of reaching configuration υ from configuration τ in one step. This gives an integer in $\{0, 1, ..., d\}$ which, divided by d, gives the probability of a transition from τ to υ.

We now have the matrix dA. Each processor activates $(k^{P(n)}I(n))^{S(n)}$ processors in $O(S(n)(P(n) + \log I(n)))$ time to be used for squaring the matrix. D squares the matrix $\lceil \log T(n) \rceil$ times in the straightforward way. Each squaring takes $O(S(n)(P(n) + \log I(n)))$ time. We now have the matrix $d^{T(n)}A^{T(n)}$. Entry (τ, υ) of $A^{T(n)}$ is the probability of reaching configuration υ from configuration τ in $T(n)$ steps. D sums the number N of ways of reaching each accepting configuration from the initial configuration. If $2N > d^{T(n)}$ (that is, if the probability of reaching an accepting configuration is greater than ½), then D accepts; otherwise, D rejects. □

8.4. Probabilistic Circuits

In this section, we relate probabilistic unbounded fan-in circuits and CRCW *prob*-PRAMs using the relationship between their deterministic counterparts (Stockmeyer and Vishkin, 1984). We find that, just as in the deterministic case, time and number of processors of a *prob*-PRAM correspond simultaneously to depth and size of a probabilistic unbounded fan-in circuit. Time and depth correspond to within a constant factor; number of processors and size correspond to within a polynomial.

A *probabilistic circuit* $PC_{n,m}$ is a circuit with n regular inputs and m random inputs.

Theorem 8.8. Let PZ be a *prob*-PRAM with time bound $T(n)$ and processor bound $P(n)$. There is a probabilistic unbounded fan-in circuit $PC_{n,P(n)T(n)}$ that simulates PZ in depth $O(T(n))$ and size $q(P(n), T(n), n)$, where $q(P, T, n)$ is bounded above by a polynomial in P, T, and n.

Proof. Theorem 2.1 states the deterministic result of Stockmeyer and Vishkin (1984). We modify the proof given by Stockmeyer and Vishkin for the simulation of a deterministic PRAM by a deterministic circuit. Assume that all probabilistic choices made by *PZ* are between two alternatives. The circuit presented by Stockmeyer and Vishkin has an identical carton of gates for each time step and each processor. Each carton is made of 13 blocks. We add a random input bit to one of these blocks: [Update-ic]. This block selects the next instruction to be executed by the simulated processor. If the instruction currently being executed calls for a random choice, then the next instruction is selected based on the value of the random bit. $PC_{n,P(n)T(n)}$ keeps the same size and depth bounds as stated in Theorem 2.1. □

Theorem 8.9. Let $PC_{n,m}$ be a probabilistic unbounded fan-in circuit of size S and depth T with n regular inputs and m random inputs. There is a *prob*-PRAM *PZ* that simulates *PC* in time $O(T + \log m)$ with $O(S + n)$ processors.

Proof. Theorem 2.2 states the deterministic result of Stockmeyer and Vishkin (1984). We modify the proof given by Stockmeyer and Vishkin for the simulation of a deterministic circuit by a deterministic PRAM. The only difference between a probabilistic circuit and a deterministic circuit is the m random inputs in the probabilistic circuit. *PZ* activates m processors in $O(\log m)$ time. Each processor simulating one of the random inputs randomly chooses a value, then writes that value to the cell corresponding to its random input.

The remainder of the simulation follows as in Stockmeyer and Vishkin (1984). □

Chapter 9. Simulation by Sequential Machines

We present here simulations of PRAMs with enhanced instruction sets by RAMs with the same instruction set through uniform, bounded fan-in circuits. We will prove that a RAM[*op*] can efficiently simulate a uniform, bounded fan-in circuit and then show that the circuits presented earlier that simulate a PRAM[*op*] meet the uniformity conditions.

9.1. Definitions

We use the following definitions relating to circuits (Ruzzo, 1981).

- A *circuit* is a directed acyclic graph, where each node (gate) with indegree $d > 0$ is labeled by the AND, OR, or NOT of d variables, and each node with with indegree 0 is labeled by "*inp*" (an *input*). Nodes with outdegree 0 are *outputs*.
- A *circuit family* C is a set $\{C_1, C_2, \cdots\}$ of circuits, where C_n has n inputs and one output. We restrict the gate numbering so that the largest gate number is $(Z(n))^{O(1)}$, where $Z(n)$ is the size of C_n. Thus the gate numbers coded in binary have length $O(\log Z(n))$.
- The family C *recognizes* $A \subseteq \{0,1\}^*$ if for each n, C_n recognizes $A^{(n)} = A \cap \{0,1\}^n$, that is, the value of C_n on input $inp_1, inp_2, ..., inp_n \in \{0,1\}$ is 1 if and only if $inp_1 \cdots inp_n \in A$.

If C_n has at most $Z(n)$ gates and depth $D(n)$, then the *size complexity* of C is $Z(n)$ and the *depth complexity* is $D(n)$. A *language* Q is of simultaneous size and depth complexity $Z(n)$ and $D(n)$ if there is a family of circuits of size complexity $Z(n)$ and depth complexity $D(n)$ that recognizes Q.

- A *bounded fan-in circuit* is a circuit where the indegree of all gates is at most 2. For each gate g in C_n, let $g(\lambda)$ denote g, $g(L)$ denote the left input to g, and $g(R)$ denote the right input to g.

• An *unbounded fan-in circuit* is a circuit where indegree is unbounded. For each gate g in C_n, let $g(\lambda)$ denote g, and let $g(p)$, $p = 0, 1, 2, \ldots$, denote the pth input to g.

• The *bounded direct connection language* of the family $C = \{C_1, C_2, \cdots\}$, L_{BDC}, is the set of strings of the form $\langle n, g, p, h \rangle$, where $n, g \in \{0, 1\}^*$, $p \in \{\lambda, L, R\}$, $h \in \{inp, \text{AND}, \text{OR}, \text{NOT}\} \cup \{0,1\}^*$ such that in C_n either (i) $p = \lambda$ and gate g is a h-gate, $h \in \{inp, \text{AND}, \text{OR}, \text{NOT}\}$, or (ii) $p \neq \lambda$ and gate $g(p)$ is numbered h, $h \in \{0,1\}^*$.

• The *unbounded direct connection language* of the family $C = \{C_1, C_2, \cdots\}$, L_{UDC}, is the set of strings of the form $\langle n, g, p, h \rangle$, where $n, g \in \{0, 1\}^*$, $p \in \{\lambda\} \cup \{0, 1, \ldots, Z(n)^{O(1)}\}$, $h \in \{inp, \text{AND}, \text{OR}, \text{NOT}\} \cup \{0,1\}^*$ such that in C_n either (i) $p = \lambda$ and gate g is a h-gate, $h \in \{inp, \text{AND}, \text{OR}, \text{NOT}\}$, or (ii) $p \neq \lambda$ and gate $g(p)$ is numbered h, $h \in \{0,1\}^*$.

Let us now introduce two new definitions of uniformity. Let $I(Z(n))$ be the concatenation of all pairs (g, h), where $g, h \in \{0, 1, \ldots, Z(n)^{O(1)}\}$.

• The family $C = \{C_1, C_2, \cdots\}$ of bounded (respectively, unbounded) fan-in circuits of size $Z(n)$ is *VM-uniform* if there is a RAM[↑,↓] that on input $I(Z(n))$ returns an output string in $O(\log Z(n))$ time indicating for each pair (g, h) whether $\langle n, g, L, h \rangle$ is in L_{BDC} and whether $\langle n, g, R, h \rangle$ is in L_{BDC} (respectively, indicating for each pair (g, h) the value of p such that $\langle n, g, p, h \rangle$ is in L_{UDC}, for $p = 0, 1, \ldots, Z(n)^{O(1)}$, or an indication that no $\langle n, g, p, h \rangle$ is in L_{UDC}). (Note: We chose the term VM-uniform because Pratt and Stockmeyer (1976) called their restricted RAM[↑,↓] a vector machine.)

• The family $C = \{C_1, C_2, \cdots\}$ of bounded (respectively, unbounded) fan-in circuits of size $Z(n)$ is *MRAM-uniform* if there is a RAM[*] that on input $I(Z(n))$ returns an output string in $O(\log Z(n))$ time indicating for each pair (g, h) whether $\langle n, g, L, h \rangle$ or $\langle n, g, R, h \rangle$is in L_{BDC} (respectively, indicating for each pair (g, h) the value of p such that

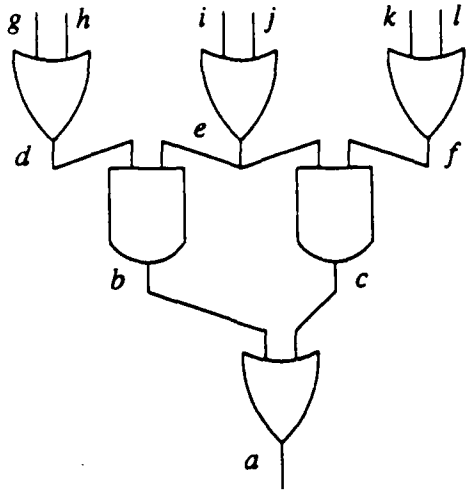
$<n, g, p, h>$ is in L_{UDC}, for $p = 0, 1, ..., Z(n)^{O(1)}$, or an indication that no $<n, g, p, h>$ is in L_{UDC}). (Note: We chose the term MRAM-uniform because Hartmanis and Simon (1974) called their RAM[*] an MRAM.)

- A gate g is at *level* j of C_n if the longest path from any circuit input to g has length j. Gate g is at *height* j of C_n if the longest path from g to the output has length j.

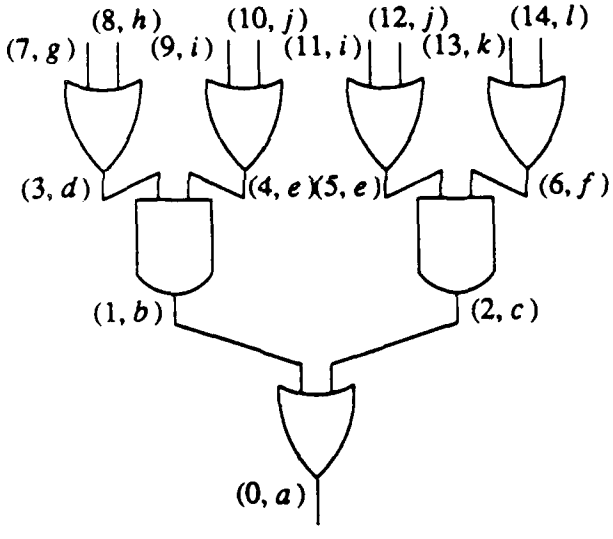
- Let C_n be a bounded fan-in circuit consisting entirely of AND, OR, and *inp* gates with depth $D(n)$. We construct the circuit $CT(C_n)$, the *circuit tree* of C_n, from C_n. Let gate a be the output gate of C_n and let a be of type $\phi \in \{AND, OR\}$ with inputs from gates b and c. Then the output gate of $CT(C_n)$ has name $(0, a)$, type ϕ, and inputs from gates named $(1, b)$ and $(2, c)$. Thus, gate $(0, a)$ is the gate at height 0 of $CT(C_n)$ and gates $(1, b)$ and $(2, c)$ are the gates at height 1 of $CT(C_n)$. Now suppose that we have constructed all gates at height j of $CT(C_n)$, and we wish to construct the gates at height $j+1$. Each gate (i, e) at height j corresponds to a gate e in C_n. If e is of type $\phi \in \{AND, OR\}$, then gate (i, e) is of type ϕ. Suppose gate e has inputs from gates f and g. Then the inputs to gate (i, e) of $CT(C_n)$ at height $j+1$ are the gates $(2i+1, f)$ and $(2i+2, g)$. If gate e is of type *inp*, that is, an input, and $j < D(n)$, then (i, e) is of type OR (if (i, e) is at an even numbered level) or type AND (if (i, e) is at an odd numbered level), and the inputs to gate (i, e) at height $j+1$ are the gates $(2i+1, e)$ and $(2i+2, e)$. If gate e is of type *inp* and $j = D(n)$, then (i, e) is of type *inp* and $CT(C_n)$ has no gates at height $j+1$ connected to gate (i, e). Figure 9.1 contains an example of a circuit tree.

- In $CT(C_n)$, define *path* (a, b) to be the path, if one exists, from gate a to gate b.

- In $CT(C_n)$, the *distance* from gate b to gate c is the length of the shortest path from b to c, if such a path exists. We order all gates at distance d from gate b according to the relation



Circuit C_6



Circuit tree $CT(C_6)$

Figure 9.1. Circuit tree example

order (e, f) such that *order* (e, f) is true if *path* (e, b) intersects *path* (f, b) at a gate which *path* (e, b) enters at the left input and *path* (f, b) enters at the right input. Gate e is the *qth ancestor of gate b at distance d* if gate e is the qth smallest gate in the ordering of the gates at distance d. We say that the smallest gate at distance d is the 0th ancestor. In the example above, gate $(10, j)$ is the 3rd ancestor of gate $(0, a)$ at distance 3. Note that the same gate in C_n can correspond to several ancestors of a gate at distance d in $CT(C_n)$.

- A *double-rail circuit* is a bounded fan-in circuit that is given as input inp_1, inp_2, ..., inp_n and their complements $\overline{inp}_1$, $\overline{inp}_2$, ..., $\overline{inp}_n$ and that contains no NOT-gates.

Note that every gate in a double-rail circuit, except the input gates, has exactly two inputs.

- A *layered circuit* is a double-rail circuit such that all gates at level i, for all odd i, are AND-gates and all gates at level i, for all even i, are OR-gates, and each input to a gate at level i is connected to an output of a gate at level $i - 1$.

Lemma 9.1.1. Let $C = \{C_1, C_2, \cdots\}$ be a VM-uniform (MRAM-uniform) family of bounded fan-in circuits of size $Z(n)$ and depth $D(n)$ recognizing language L. There exists a VM-uniform (MRAM-uniform) family of bounded fan-in, double-rail circuits $E = \{E_1, E_2, \cdots\}$ of size $O(Z(n))$ and depth $D(n)$ recognizing language L.

Proof. Fix an input size n. We construct E_n from C_n. Suppose that for each input inp_i we have its complement, $\overline{inp}_i$. For each gate g of type $y \in \{inp, \text{AND}, \text{OR}\}$ in C_n, circuit E_n has a gate g of type y. E_n also has a gate g': if $y = \text{AND}$, then g' is of type OR; if $y = \text{OR}$, then g' is of type AND; and if $y = inp$ and gate g is input inp_i, then gate g' is of type inp and is input $\overline{inp}_i$. Suppose h is an input to gate g in C_n. If h is of type $y \in \{inp, \text{AND}, \text{OR}\}$, then in

E_n, gate h is also an input to gate g and gate h' is an input to gate g'. If h is of type NOT with input f, then in E_n, gate f' is an input to g and gate f is an input to g'.

Thus, E is a VM-uniform (MRAM-uniform) family of double-rail circuits with size $O(Z(n))$ and depth $D(n)$ recognizing language L. □

Lemma 9.1.2. Let $E = \{E_1, E_2, \cdots\}$ be a family of VM-uniform (MRAM-uniform), bounded fan-in, double-rail circuits of size $Z(n)$ and depth $D(n)$ recognizing language L. There exists a family of VM-uniform (MRAM-uniform), bounded fan-in, layered circuits $F = \{F_1, F_2, \cdots\}$ of size $O(Z(n))$ and depth $O(D(n))$ recognizing language L.

Proof. Fix an input size n. We construct F_n from E_n. Suppose j is odd and that we have constructed $j-1$ levels of F_n from the first $i-1$ levels of E_n. We construct level j of F_n. First assume that all inputs to gates in level i of E_n are outputs from level $i-1$. If all gates in level i of E_n are AND-gates, then level j of F_n is identical to level i of E_n, and we move on to construct level $j+1$ of F_n. Otherwise, for each AND-gate in level i of E_n, we place an AND-gate in the corresponding place in level j of F_n. For each of these AND-gates h in F_n in level j, there is an OR-gate g in level $j+1$ with two inputs h. For each OR-gate in level i of E_n, we place an OR-gate in the corresponding place in level $j+1$ of F_n. For each input f of these gates, we place an AND-gate in level j with two inputs f. Then we move on to construct level $j+2$ of F_n. If j is even, we construct level j of F_n similarly.

Now we remove the assumption that all inputs to gates in level i of E_n are outputs from level $i-1$. Construct F_n as described above with one exception. If an input to a gate g_E in E_n at level i is the output from gate h_E at level i', $i' \neq i-1$, then do not directly connect the output from the corresponding gate h_F to an input of g_F in F_n. Instead, between gates h_F and g_F insert alternating AND- and OR-gates, with AND-gates at odd levels and OR-gates at

even levels, and let both inputs to a gate at level k be connected to the output of the newly inserted gate at level $k-1$.

Thus, F is a VM-uniform (MRAM-uniform) family of layered circuits with size $O(Z(n))$ and depth $O(D(n))$ recognizing language L. □

9.2. Simulation of VM-Uniform Circuit by RAM[↑,↓]

In this section, we restructure a VM-uniform, bounded fan-in circuit, then simulate the restructured circuit on a RAM[↑,↓]. The RAM[↑,↓] will operate on a circuit tree rather than the original circuit because the RAM[↑,↓] can very easily run a circuit tree on an input, once the input bits are properly placed. The RAM[↑,↓] will also split the circuit into slices of depth $O(\log Z(n))$. In this way, the RAM[↑,↓] will balance the time to generate the circuit with the time to run the circuit.

Let $C = \{C_1, C_2, \cdots\}$ be a VM-uniform family of bounded fan-in circuits of size $Z(n)$ and depth $D(n)$ recognizing language L. For an integer x, let $\#x$ denote its two's complement representation; the number of bits in the two's complement representation will be clear from the context. We now describe how a RAM[↑,↓] can simulate C.

Simulation. Fix an input length n. Circuit C_n has size $Z(n)$ and depth $D(n)$. We construct a RAM[↑,↓] R that recognizes $L^{(n)}$ in time $O(D(n) + \log Z(n) \operatorname{loglog} Z(n))$.

By Lemmas 9.1.1 and 9.1.2, we construct a layered circuit F_n from C_n with size $O(Z(n))$ and depth $O(D(n))$ that recognizes language $L^{(n)}$. Machine R simulates C_n via F_n. For simplicity, let us say that F_n has depth $D(n)$ and size $Z(n)$, and that all gates of F_n are numbered from $\{0, 1, \ldots, Z(n)-1\}$.

Let us first outline the simulation.

Stage 1. R generates a $Z(n) \times Z(n)$ ancestor matrix A in which each entry (g, h) indicates whether gate h is an input of gate g in F_n.

Stage 2. R operates loglog $Z(n)$ times on matrix A, obtaining matrix $A_{\log Z(n)}$.

Stage 3. R extracts from $A_{\log Z(n)}$ a description of the circuit in slices of depth $O(\log Z(n))$ and their circuit trees.

Stage 4. R runs each slice consecutively on the input.

Stage 1: Computation of ancestor matrix. Each entry in ancestor matrix A is a bit vector $Z(n)$ bits long. Entry (g, h) has a 1 in bit position 0 (1) if gate h is the left (right) input to gate g; otherwise, bit position 0 (1) holds 0. All other bit positions hold 0. Let $A_1 = A$. In general, R operates on A_i to produce A_{2i}. After loglog $Z(n)$ operations, $A_{\log Z(n)}(g, h)$ holds 1 in each bit position j if gate h is the jth ancestor of gate g at distance log $Z(n)$, 0 otherwise.

R will first write all pairs (g, h), where $g, h \in \{0, 1, ..., Z(n)\text{-}1\}$, concatenated in a single register in $O(\log Z(n))$ time. We view a register as the concatenation of $Z^2(n)$ *slots*, each slot $Z(n)$ bit positions long. Pair (g, h) is written in slot $gZ(n) + h$ with the least significant bit of $\#g$ in the 0th bit position in the slot and the least significant bit of $\#h$ in the $Z(n)$/2th bit position in the slot. R constructs the first component of every pair one bit position at a time, then the second component of every pair one bit position at a time.

We now describe how R builds the first component of each pair; R builds the second component similarly. We build the first component as a bit vector with $\#g$ in each of slots $gZ(n),...,(g+1)Z(n) - 1$, for each g, $0 \le g \le Z(n)-1$. Let υ denote this bit vector.

R first constructs a bit vector ξ in which each slot $gZ(n)$, $0 \le g \le Z(n) - 1$, holds $\#g$. Let $q = \log Z(n)$. Each integer g is q bits long. R constructs ξ in q phases, generating one bit

position at a time for all g. Let $mask_i$ denote the bit vector where $\#mask_i$ has a 1 in the ith bit position of each slot and 0's elsewhere. Let $S_i = \xi \wedge mask_i$; that is, in the ith bit position of each slot, $\#S_i$ equals $\#\xi$ and is 0's elsewhere. Thus, $\xi = \bigvee_{i=0}^{q-i} S_i$. In phase i, R constructs S_{q-i}, using $O(q)$ time in phase 1 and $O(1)$ time in every other phase.

To build S_{q-1}, start with

$$t_1 \leftarrow 1 \uparrow [(Z(n))(Z(n)/2) + q - 1],$$

then for $j = 1, \ldots, q-1$, execute the following:

$$u_j \leftarrow t_j \uparrow (Z(n) \cdot 2^{j-1})$$

$$t_{j+1} \leftarrow t_j \vee u_j.$$

Finally, $S_{q-1} = t_q$, and R writes S_{q-1} in r_1 and r_2.

At the start of phase i, register r_1 contains $S_{q-1} \vee S_{q-2} \vee \cdots \vee S_{q-i+1}$, and register r_2 contains S_{q-i+1}. R constructs S_{q-i} from S_{q-i+1} as follows:

$r_3 \leftarrow r_2 \downarrow (2^{q-i})(Z^2(n))$ (* $Z(n)$ is the slot size and slots holding g's are $Z(n)$ apart *)

$r_4 \leftarrow r_2 \wedge r_3$ (* half of the 1's in $\#rcon(2)$ *)

$r_5 \leftarrow r_2 \oplus r_4$ (* the other half of the 1's in $\#rcon(2)$ *)

$r_6 \leftarrow r_4 \downarrow (2^{q-i})(Z^2(n))$

$r_2 \leftarrow r_5 \vee r_6$

$r_1 \leftarrow r_1 \vee r_2$

and r_1 holds $S_{q-1} \vee \cdots \vee S_{q-i+1} \vee S_{q-i}$, and register r_2 holds S_{q-i}.

Each phase after the first takes $O(1)$ time, and R builds ξ in $O(q) = O(\log Z(n))$ steps.

R next builds υ from ξ by filling in the empty slots in $\log Z(n)$ phases. Let $t_1 = \xi$. In phase i, R takes the output t_i of phase $i-1$ and computes

$$u_i \leftarrow t_i \uparrow Z(n) \cdot 2^i$$

$$t_{i+1} \leftarrow u_i \vee t_i.$$

We set $\upsilon = t_{\log Z(n)+1}$. In this manner, R builds the first component of each pair in $O(\log Z(n))$ time.

R builds the second component of each pair similarly. R now has a register containing the concatenation of all pairs (g, h), $0 \le g, h \le Z(n) - 1$. For each pair (g, h), R determines simultaneously whether gate h is an input to gate g. If gate h is the left (right) input to gate g, then R writes a 1 in the first (second) position in the slot. By VM-uniformity, this process takes $O(\log Z(n))$ time. The resulting bit vector constitutes the ancestor matrix A.

Stage 2: Computation of distance $\log Z(n)$ *ancestor matrix.* Pratt and Stockmeyer (1976) proved that given two $z \times z$ Boolean matrices A and B, a RAM[$\uparrow$,$\downarrow$] can compute their Boolean product G, defined by

$$G(i, j) = \bigvee_k (A(i, k) \wedge B(k, j)),$$

in $O(\log z)$ time. The RAM[$\uparrow$,$\downarrow$] performs the AND of all triples i, j, k in one step and the OR in $O(\log z)$ steps. Let θ_d be a function, specified below, with two bit vectors as inputs and one bit vector as output. Given two $z \times z$ matrices A and B whose elements are bit vectors m bits long, let us define the function $H_d(A, B) = G$, where

$$G(i, j) = \bigvee_k \theta_d(A(i, k), B(k, j)).$$

We prove that a RAM[$\uparrow$,$\downarrow$] can compute the matrix $G = H_d(A, B)$ in $O(\log z + \log m)$ time. The RAM[$\uparrow$,$\downarrow$] performs θ_d on all triples i, j, k in $O(\log z + \log m)$ steps and the OR in $O(\log z)$ steps. Since we have the matrix multiplication algorithm of Pratt and Stockmeyer, we prove that a RAM[$\uparrow$,$\downarrow$] can compute $\theta_d(A(i, k), B(k, j))$ in $O(\log m)$ time to establish this bound. In our case, $m = z = Z(n)$.

Suppose R has operated $\log d$ times on A. Call the resulting matrix A_d. A 1 in bit position i of $A_d(f, g)$ indicates that gate g is the ith ancestor of gate f at distance d. We want $\theta_d(A_d(f, g), A_d(g, h))$ to return a bit vector $A_{2d}(f, h)$ with a 1 in bit position i if gate h is the ith ancestor of gate f at distance $2d$, and 0 in bit position i otherwise.

Assume m is a power of 2. Let $x = 2^d$. Let α and β be integers x bits long: $\#\alpha = \alpha_{x-1} \cdots \alpha_1 \alpha_0$ and $\#\beta = \beta_{x-1} \cdots \beta_1 \beta_0$. We define the function $\theta_d(\alpha, \beta)$ so that for each 1 in bit position a in $\#\alpha$ and each 1 in bit position b in $\#\beta$, in the result $\gamma = \theta_d(\alpha, \beta)$, $\#\gamma$ has a 1 in position $a^* 2^d + b$. If either α or β is 0, then γ is 0.

Before describing how R performs θ_d, we define two functions *SPREAD* and *FILL* that R will use to perform θ_d. The function *SPREAD* $(\#\alpha, y)$ returns the bit vector $\#\alpha' = \alpha_{x-1}0 \cdots 0\alpha_{x-2}0 \cdots 0\alpha_1 0 \cdots 0\alpha_0$ in which α_{i+1} is y bit positions away from α_i, separated by 0's, for all $0 \le i \le x - 2$.

Lemma 9.1.3. For any y, R can perform *SPREAD* $(\#\alpha, y)$ in time $O(\log x)$, where $\#\alpha$ is x bits long.

Proof. R performs *SPREAD* in $O(\log x)$ phases of mask and shift operations. Note that the subscript of each bit in $\#\alpha$ specifies its position. In phase i, R uses $mask_i$ to mask away all bits of $\#\alpha$ whose subscript has a 0 in the $(\log x - i)$th position. R then shifts the remainder of the string.

In particular, R creates $mask_1$ in three steps: $mask_1 \leftarrow 1 \uparrow (x/2)$; $mask_1 \leftarrow mask_1 - 1$; $mask_1 \leftarrow mask_1 \uparrow (x/2)$. Thus, $\#mask_1$ has 1's in its $x/2$ most significant positions and 0's in the remaining $x/2$ bit positions. (We can readily divide by powers of 2 using the right shift operation and multiply by powers of 2 using the left shift operation.) In general, $\#mask_i$ has a 1 in every position p such that p has a 1 in the $(\log x - i)$th bit position.

Let $temp_1 = \alpha$. We let $temp_i$ hold intermediate results as R spreads the bits in α to get α'.

Given $temp_i$ and $mask_i$, R constructs $temp_{i+1}$ and $mask_{i+1}$ in phase i as follows.

$temp_i(1) \leftarrow temp_i \wedge mask_i$ (* mask away bits of $\#\alpha$ whose subscript has 0 in the $(\log x - i)$th bit position *)

$temp_i(2) \leftarrow temp_i \oplus temp_i(1)$ (* the bits of $\#\alpha$ masked away in the previous step *)

$temp_i(1) \leftarrow temp_i(1) \uparrow ((x/2^i)*(y-1))$ (* spread the unmasked bits *)

$temp_{i+1} \leftarrow temp_i(1) \vee temp_i(2)$ (* combine with masked bits *)

$mask_i(1) \leftarrow mask_i \downarrow (x/2^{i+1})$

$mask_i(2) \leftarrow mask_i(1) \wedge mask_i$

$mask_i(3) \leftarrow mask_i(1) \oplus mask_i(2)$ (* separate bits of $mask_i$ into two groups *)

$mask_i(4) \leftarrow mask_i(2) \uparrow [(x/2^i)*(y-1)+(x/2^{i+1})]$

$mask_{i+1} \leftarrow mask_i(3) \vee mask_i(4)$

In $O(\log x)$ phases, each taking a constant amount of time, R performs *SPREAD* $(\#\alpha, y)$. □

Let $\alpha^s = SPREAD(\#\alpha, y)$. The function $FILL(\alpha^s, y)$ returns the value α^f, where $\#\alpha^f = \alpha_{x-1}\alpha_{x-1} \cdots \alpha_{x-1}\alpha_{x-2} \cdots \alpha_1\alpha_1 \cdots \alpha_1\alpha_0\alpha_0 \cdots \alpha_0$, in which positions iy, ..., $(i+1)y-1$ have value α_i. Assume y is a power of 2. (Note: One may think of α^s as α spread out and α^f as α^s filled in.)

Lemma 9.1.4. R can perform $FILL(\alpha^s, y)$ in time $O(\log y)$.

Proof. R performs $FILL(\alpha^s, y)$ in $O(\log y)$ phases of shift and OR operations. Let $\alpha^f(i)$ represent the outcome of phase i.

$$temp \leftarrow \alpha^f(i) \uparrow 2^{i-1}$$

$$\alpha^f(i+1) \leftarrow \alpha^f(i) \vee temp$$

Then $FILL(\alpha^s, y) = \alpha^f = \alpha^f(\log y)$. Each phase takes constant time, so R performs $FILL(\alpha^s, y)$ in $O(\log y)$ time. □

Now we describe how a RAM[↑,↓] R computes $\theta_d(\alpha, \beta)$ in $O(\log x) = O(\log m)$ steps. R first computes $\alpha^s = SPREAD(\#\alpha, 2^d)$ in $O(\log x)$ time. Each 1 in position i in $\#\alpha$ produces a 1 in position $i \cdot 2^d$ of $\#\alpha^s$. R concatenates $x = 2^d$ copies, each m bits long, of $\#\alpha^s$ in $O(\log x)$ time. Let *squid* denote the value of this concatenation. R then computes $\beta^s = SPREAD(\#\beta, m)$ in $O(\log x)$ time; this aligns a bit of $\#\beta$ with each copy of $\#\alpha^s$ in *#squid*.

R computes $\beta^f = FILL(\beta^s, m)$. R then performs $squid \leftarrow squid \wedge \beta^f$, which blocks out each copy of $\#\alpha^s$ in *#squid* that corresponds to a 0 in $\#\beta$.

Next we explain how for each nonzero bit β_j of $\#\beta$, R shifts the jth copy of $\#\alpha^s$ to the left by j bits in $O(\log x)$ phases of mask and shift operations. In phase i, R masks away all copies of $\#\alpha^s$ corresponding to nonzero bits β_j for which the $(\log x - i)$th position of $\#j$ is 0, then shifts the remainder of the vector.

R creates $mask_1$ as follows: $mask_1 \leftarrow 1 \uparrow (mx/2)$; $mask_1 \leftarrow mask_1 - 1$; $mask_1 \leftarrow mask_1 \uparrow (mx/2)$. Thus, $\#mask_1$ has 1's in its $mx/2$ most significant positions and 0's in the remaining $mx/2$ bit positions.

Let $temp_1 = squid$.

Given $temp_i$ and $mask_i$, R constructs $temp_{i+1}$ and $mask_{i+1}$ in phase i as follows.

$temp_i(1) \leftarrow temp_i \wedge mask_i$

$temp_i(2) \leftarrow temp_i \oplus temp_i(1)$ (* split bits of $temp_i$ into two sets *)

$temp_i(1) \leftarrow temp_i(1) \uparrow (2^{\log x - i})$ (* shift one set *)

$temp_{i+1} \leftarrow temp_i(1) \vee temp_i(2)$ (* recombine *)

$mask_i(1) \leftarrow mask_i \downarrow (m/2^{i+1})$

$mask_i(2) \leftarrow mask_i(1) \wedge mask_i$

$mask_i(3) \leftarrow mask_i(1) \oplus mask_i(2)$ (* split bits of $mask_i$ into two sets *)

$mask_i(4) \leftarrow mask_i(2) \uparrow (x/2^i)$ (* shift one set *)

$mask_{i+1} \leftarrow mask_i(3) \vee mask_i(4)$ (* recombine *)

Let a block of size m of $\#\gamma = \gamma_x \cdots \gamma_0$ be a set of bits $\gamma_{(j+1)m-1} \cdots \gamma_{jm}$. Finally, R ORs together all blocks of size m in $O(\log m)$ steps. (Formerly, each block of size m was a copy of $\#\alpha^s$; now some have been shifted.) The resulting bit vector is $\theta_d(\alpha, \beta)$. R has computed $\theta_d(\alpha, \beta)$ in $O(\log m)$ steps.

Recall that we defined $H_d(A, B) = G$, where

$$G(i, j) = \bigvee_k \theta_d(A(i, k), B(k, j)).$$

To perform $H_d(A, B)$ on two $z \times z$ matrices A and B in $O(\log z + \log m)$ time, we must show that we can perform $\theta_d(A(i,k), B(k,j))$ in $O(\log z + \log m)$ time when the matrices A and B are given as bit vectors. We simply allow enough space between elements in the bit vectors A and B so that operations on adjacent pairs do not interfere with each other, and we can generate all masks and perform all operations in $O(\log z + \log m)$ time.

Given the $Z(n) \times Z(n)$ ancestor matrix $A_1 = A$ with entries $Z(n)$ bits long, R computes $A_2 = H_1(A_1, A_1)$, then $A_{2j} = H_j(A_j, A_j)$, for each $j = 1, 2, 4, \cdots, \log Z(n) - 1$. Let $G = A_{\log Z(n)}$. Each H_d operation takes time $O(\log Z(n))$ to compute, and R executes H_d for loglog $Z(n)$ values of d. Hence, R computes G in $O(\log Z(n) \text{ loglog } Z(n))$ time.

Stage 3: Extraction of slices and circuit trees. We partition F_n into $D(n) / \log Z(n)$ *slices* of depth $\log Z(n)$ each. Let $v = D(n) / \log Z(n)$. The jth slice comprises levels $j \log Z(n)$, ..., $(j+1) \log Z(n) - 1$, for $0 \le j \le v - 1$. Let $\Sigma(j)$ denote the jth slice.

R will extract circuit tree descriptions of each slice from matrix G, starting with $\Sigma(v-1)$.

We introduce a simple procedure *COLLAPSE*, which R will use to extract information from matrix G. Procedure *COLLAPSE* (α, z) takes as input the value α, where $\#\alpha = \alpha_{z^2-1} \cdots \alpha_1 \alpha_0$, and returns the value β, where $\#\beta = \beta_{z^2-1} \cdots \beta_1 \beta_0$, and bits $\beta_{kz} = \bigvee_{j=0}^{z-1} \alpha_{kz+j}$, for $0 \le k \le z - 1$, and $\beta_i = 0$ if $i \ne kz$.

Lemma 9.1.5. R can perform *COLLAPSE* (α, z) in $O(\log z)$ time.

Proof. Let $temp_1 = \alpha$. For $i = 0, ..., \log z - 1$, R performs the following steps:

$$temp_i(1) \leftarrow temp_i \downarrow 2^i$$

$$temp_{i+1} \leftarrow temp_i \vee temp_i(1).$$

Let ψ denote $temp_{\log z}$. At this point $\#\psi = \psi_{z^2-1} \cdots \psi_1 \psi_0$, where $\psi_{kz} = \bigvee_{j=0}^{z-1} \alpha_{kz+j}$. Now we mask away all bits ψ_i for $i \ne kz$. This takes $O(\log z)$ time, and the result is β. □

Matrix G is stored in a single register in row major order. We view the contents of this register as both a matrix of discrete elements and as a single bit string. We call the portion of a matrix comprising one row a *box*. We call the portion of a box containing one element of a row a *slot*.

To extract $CT(\Sigma(i))$, R will isolate the portion of matrix G that describes the circuit trees for each output from slice $\Sigma(i)$. We will call this matrix a *slice matrix*. Let $S(i)$ denote the slice matrix for $\Sigma(i)$. For each gate g at the output of $\Sigma(i)$, $S(i)$ specifies each ancestor of gate g at the top of $\Sigma(i)$.

Let *out* denote the name of the output gate of F_n. Nonzero entries in row *out* of G correspond to the ancestors of gate *out* at distance $\log Z(n)$; that is, the gates at the boundary between $\Sigma(v-1)$ and $\Sigma(v-2)$. To extract $CT(\Sigma(v-1))$, R masks away all but row *out* of G. Let $S(v-1)$ denote this value.

In general, assume that we have $S(i+1)$, and we want to compute $S(i)$. First, R computes the OR of all boxes of $S(i+1)$. Let $\eta(i)$ denote this value. R computes $\alpha(i) = COLLAPSE(\eta(i), Z(n))$ in $O(\log Z(n))$ time. Bit $jZ^2(n) + kZ(n)$, $0 \le j, k \le Z(n) - 1$, of $\#\alpha(i)$ is 0 (1) if slot k of box j of $S(i+1)$ contains all 0's (at least one 1). Let $\alpha_b(i)$ denote bit b of $\#\alpha(i)$. We will use $\alpha(i)$ to select the rows of G that correspond to nonzero slots of $S(i+1)$. Next, R computes $\sigma(i) = SPREAD(\#\alpha(i), Z(n))$ in $O(\log Z(n))$ steps. This leaves bit $\alpha_{kZ(n)}(i)$ at position $kZ^2(n)$ of $\#\sigma(i)$; that is, it aligns the bit of $\#\alpha(i)$ indicating whether or not slot m contains a 1 with box m of G. Now, R computes $\phi(i) = FILL(\sigma(i), Z^2(n))$, and $\#\phi(i)$ has 1's in the boxes of G that correspond to ancestors of *out* at distance $\log Z(n)$.

R computes $S(i) \leftarrow \phi(i) \wedge G$. Thus, the boxes of G that correspond to slots of $S(i+1)$ that contain all 0's are masked away in $S(i)$. Each nonzero slot of $S(i)$ indicates a gate at the top boundary of $\Sigma(i)$.

Stage 4: Running the slices on the input. At this point, for $0 \le i \le v - 1$, R has computed $S(i)$. Each $S(i)$ contains a description of $CT(\Sigma(i))$. (Note: $CT(\Sigma(i))$ is a collection of circuit trees, one for each output of $\Sigma(i)$.) In Stage 4, R runs each $CT(\Sigma(i))$ in sequence. R

begins by manipulating the input ω to be in the form necessary to run on $CT(\Sigma(0))$. The input ω to F_n is $2n$ bits long (n input bits and their complements).

We describe how R runs the circuit by slices. We must take the output $\psi(i)$ from $\Sigma(i)$ and convert it into the form needed for the input to $CT(\Sigma(i+1))$. We let $\omega(i)$ denote this input.

We must initially consider input ω as a special case. Without loss of generality assume that the input gates are numbered 1,..., $2n$. We view ω as padded with 0's to be $Z(n)$ bits long. R computes $\mu(0)$ $SPREAD(\#\omega, Z(n))$. The bits of ω are $Z(n)$ bits apart in $\mu(0)$, one per slot in a single box.

We now define a function *COMPRESS*, the inverse of *SPREAD*. The function *COMPRESS*(β, y), where $\#\beta = \beta_{xy-1} \cdots \beta_1\beta_0$, returns the value α, where $\#\alpha = \beta_{(x-1)y}\beta_{(x-2)y} \cdots \beta_y\beta_0$, in $O(\log x)$ time. In general, we have $S(i)$ and $\psi(i-1)$. The output $\psi(i-1)$ from a slice $\Sigma(i-1)$ is in the form of isolated bits, one for each box corresponding to an output from $\Sigma(i-1)$. R computes $\mu(i-1) = COMPRESS(\psi(i-1), Z(n))$, then builds $\psi^f(i-1) = FILL(\mu(i-1), Z(n))$ in $O(\log Z(n))$ time. The result $\mu(i-1)$ has $Z(n)$ output bits from $\psi(i-1)$. These are $Z(n)$ bits apart, one per slot in a single box. Now R concatenates $Z(n)$ copies of $\psi^f(i-1)$; call this $\psi^c(i-1)$. Each element is $Z^2(n)$ bits long, the length of a box. R computes $S(i)' = S(i) \wedge \psi^c(i-1)$; hence, $\#S(i)'$ has a 1 in position $jZ^2(n) + kZ(n) + l$ if gate j is at the bottom of $\Sigma(i)$, gate k is the lth ancestor of j at the top of $\Sigma(i)$, and the input to gate k is a 1. R ORs all slots in each box together in $O(\log Z(n))$ steps, producing bit vector $\omega(i)$. By our construction, $\omega(i)$ is the input to $CT(\Sigma(i))$. Recall that $CT(\Sigma(i))$ consists of alternating layers of AND and OR gates. We run $CT(\Sigma(i))$ on input $\omega(i)$ in $O(\log Z(n))$ steps. Let $temp_0 = \omega(i)$. R performs the following for $m = 0, 1,$ ..., $\log Z(n) - 1$.

$$temp_m(1) \leftarrow temp_m \downarrow 2^m$$

$$temp_{m+1} \leftarrow \begin{cases} temp_m \wedge temp_m(1), \text{ if } m \text{ is even} \\ temp_m \vee temp_m(1), \text{ if } m \text{ is odd} \end{cases}$$

Let $\psi(i) = temp_{\log Z(n)}$. At most $O(Z(n))$ bits of $\#\psi(i)$ are the output values of $CT(\Sigma(i))$ on input $\omega(i)$.

It takes time $O(\log Z(n))$ to fix the output from one slice to go into another slice and time $O(\log Z(n))$ to run a slice. Since there are $D(n) / \log Z(n)$ slices, it takes time $O(D(n))$ to run a circuit on the input, given the distance $\log Z(n)$ ancestor matrix G.

Theorem 9.1. Let $C = \{C_1, C_2, \cdots\}$ be a family of VM-uniform, bounded fan-in circuits of size $Z(n)$ and depth $D(n)$ recognizing language L. There exists a RAM[$\uparrow$,$\downarrow$] R that recognizes L in time $O(D(n) + \log Z(n) \text{ loglog } Z(n))$.

Proof. We construct R by the method described above. For fixed n, R simulates C_n via F_n in $O(\log Z(n) \text{ loglog } Z(n))$ time to create matrix G, then $O(D(n))$ time to run F_n on input ω, given G. Thus, the overall time is $O(D(n) + \log Z(n) \text{ loglog } Z(n))$ steps. □

9.3. Simulation of PRAM[$\uparrow$,$\downarrow$] by RAM[$\uparrow$,$\downarrow$]

Using Theorem 9.1, we now simulate a PRAM[$\uparrow$,$\downarrow$] by a RAM[$\uparrow$,$\downarrow$]. Recall that we simulated a PRAM[$\uparrow$,$\downarrow$] by a family of log-space uniform unbounded fan-in circuits UC according to the simulation by Stockmeyer and Vishkin (1984) (Lemma 6.2.2), then simulated this by a family of log-space uniform bounded fan-in circuits BC (Lemma 6.2.3). In this manner, we showed that a family BC of bounded fan-in circuits of depth $O(T^3(n))$ and size $O(T(n)2^{T^2(n)})$ can simulate time $T(n)$ on a PRAM[$\uparrow$,$\downarrow$]. We need only establish that BC is VM-uniform to give a $O(T^3(n))$ time simulation of a PRAM[$\uparrow$,$\downarrow$] by a RAM[$\uparrow$,$\downarrow$].

Definition. A PRAM is *uniform* if all processors execute the same program.

Lemma 9.2.1. Let $C = \{C_1, C_2, ...\}$ be the family of unbounded fan-in circuits described by Stockmeyer and Vishkin (1984) that simulates a uniform PRAM (Theorem 2.1). C is VM-uniform.

Proof. Stockmeyer and Vishkin present the simulation of a nonuniform PRAM by a nonuniform family of circuits. Since we study a uniform PRAM, the program size is constant, and the simulating family of circuits is log-space uniform.

Fix a uniform PRAM Y and an input size n. The simulating circuit C_n comprises $T(n)$ identical *time slices*. Each time slice corresponds to a time step of Y. Each time slice comprises $P(n)$ *cartons* of gates, one for each processor, and a *block* of gates, [Update-Common], handling updates to common memory Each carton comprises 13 *blocks* of gates handling various functions as indicated by their names: [Compute-Operands], [Add], [Sub], [Local-Read], [Common-Read], [=-Compare], [<-Compare], [Compute-Address-of-Result], [Select-Result], [Update-Instruction-Counter], [Local-Change?], [Common-Change?], and [Update-w-Bits-of-Local-Triples]. The size of each time slice of C_n is $O(P(n)[T(n)(n+T(n)) + (n+T(n))^3 + (n+T(n))(n+P(n)T(n))])$, and the total size of C_n is $T(n)$ times this amount.

The general form of a gate name is specified in Figure 9.2.

Let $Z(n)$ denote the size of C_n. It is clear from the description of the blocks given by Stockmeyer and Vishkin that each block is VM-uniform and that the interconnections between blocks are regular. Thus, to prove that C is VM-uniform, we present an algorithm that a RAM[$\uparrow,\downarrow$] R can run to test the connectivity of all pairs of gates in $O(\log Z(n))$ time.

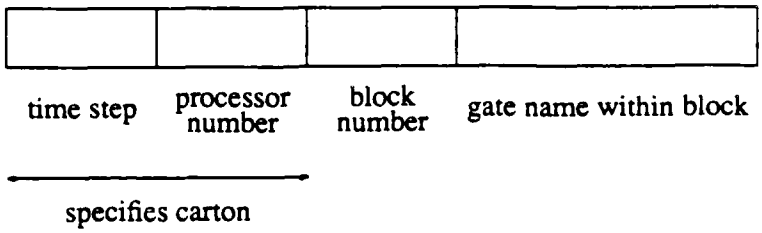


Figure 9.2. Gate name in C_n

Let g denote a gate name. Let *slot A* denote the portion of $\#g$ specifying the time step. Let *slot B* denote the portion of $\#g$ specifying the processor number. Let *slot C* denote the portion of $\#g$ specifying the block number. Let *slot D* denote the portion of $\#g$ specifying the gate name within the block. Let $scon_i(g)$ denote the contents of slot i of $\#g$.

The input I is the concatenation of all pairs (g, h), where $g, h \in \{0, 1, ..., Z(n)^{O(1)}\}$. Let the portion of that register holding the jth pair be called $pair(j)$. R initially builds four masks: $mask_A$, $mask_B$, $mask_C$, and $mask_D$ in $O(\log Z(n))$ time, such that $\#mask_i$ has 1's in slot i of every pair.

In the algorithm below, R compares parts of $\#g$ and $\#h$ for all pairs (g, h) simultaneously. R separates the pairs for which the comparison is true from the pairs for which the comparison is false by building an appropriate mask in time $O(\log Z(n))$. For all pairs (g, h):

1. Using $mask_A$, test whether $scon_A(g) = scon_A(h)$.
Mask away the unequal pairs in I. Call the resulting value I_A. Specifically, $\#I_A$ comprises pairs $(\#g, \#h)$ for which $scon_A(g) = scon_A(h)$, and 0's at the positions of pairs (g', h') for which $scon_A(g') \neq scon_A(h')$.
Mask away the equal pairs in I. Call the resulting value $I_{\bar{A}}$.

2. Test whether $scon_A(g) = 1 + scon_A(h)$ in $I_{\bar{A}}$. Mask away those pairs for which

$scon_A(g) \neq 1 + scon_A(h)$. Call the resulting value I_{A+}. Mark the pairs (g, h) for which $scon_A(g) \neq 1 + scon_A(h)$ with a 0 to indicate that h is not an input to g. (Gate h is neither in the same time slice as g nor in the preceding time slice.)

3. Using $mask_B$, test whether $scon_B(g) = scon_B(h)$ in I_A.
Mask away the unequal pairs. Call the resulting value I_B.
Mask away the equal pairs. Call the resulting value $I_{\bar{B}}$.

4. Using $mask_C$, test whether $scon_C(g) = scon_C(h)$ in I_B.
Mask away the unequal pairs. Call the resulting value I_C
Mask away the equal pairs. Call the resulting value $I_{\bar{C}}$.

5. Using $mask_D$ on I_C, isolate $scon_D(g)$ and generate the names of the leftmost and rightmost inputs to gate g.

6. Test whether $scon_D(h)$ in I_C is contained in the range specified by the names of g's leftmost and rightmost inputs.
If so, then mark pair (g, h) with a 1 to indicate that h is an input to g.
If not, then mark pair (g, h) with a 0 to indicate that h is not an input to g.

7. (slot C not equal) Using $mask_B$ on $I_{\bar{C}}$, test whether the block containing h is an input to the block containing g.
If not, then mark pair (g, h) with a 0.
Mask away the pairs for which the test is false. Call the resulting value $I_{B\bar{C}}$.

8. Using $mask_D$ on $I_{B\bar{C}}$, test whether h is an input to g between blocks. (That is, test whether h is an input to g and where g and h are in different blocks.)
If so, then mark pair (g, h) with a 1.
If not, then mark pair (g, h) with a 0.

9. (slot B not equal) Using $mask_B$ on $I_{\bar{B}}$, test whether g is in the block [Update-Common].
If not, then mark pair (g, h) with a 0. (Gates g and h are in cartons belonging to different processors.)
If so, then test whether h is in a block that feeds into [Update-Common]. If not, then mark pair (g, h) with a 0. Mask away those pairs in $I_{\bar{B}}$ that fail the test. Call the resulting value $I_{B\bar{B}}$.

10. ($scon_A(g) = 1 + scon_A(h)$) Using $mask_B$ on I_{A+}, test whether outputs from the block containing h are inputs to the block containing g.
If not, then mark pair (g, h) with a 0.
Mask away those pairs in I_{A+} that fail the test. Call the resulting value I_{BA+}, and OR this with $I_{B\bar{B}}$. Call the resulting value I_{B+}.

11. Using $mask_D$ on I_{B+}, test whether h is an input to g between blocks. (That is, test whether h is an input to g and where g and h are in different blocks.)

If so, then mark pair (g, h) with a 1.
If not, then mark pair (g, h) with a 0.

Note that the algorithm has a constant number of steps, with no loops. Also note that each step can be executed in time $O(\log Z(n))$. Thus, C is VM-uniform. □

Let $BC' = \{BC'_1, BC'_2, ...\}$ be the family of bounded fan-in circuits that simulates the family C of unbounded fan-in circuits described by Stockmeyer and Vishkin (1984) (Theorem 2.1). The depth of BC'_n is $O(T(n)(\log P(n)T(n)))$, and the size is $O(P(n)T(n)[T(n)(n+T(n)) + (n+T(n))^3 + (n+T(n))(n+P(n)T(n))])$.

Lemma 9.2.2. BC' is VM-uniform.

Proof. Fix an input length n. By Lemma 9.2.1, C is VM-uniform. The fan-in of any gate in C_n is at most $O(nP(n)(n + P(n)T(n)))$. We construct BC' from C by replacing each gate of C_n with fan-in f by a tree of gates of depth $\log f$. Thus, each gate in C_n can be simulated by a tree of gates in BC'_n of depth at most $O(\log P(n)T(n))$. Hence, BC'_n is VM-uniform. □

Theorem 9.2. For all $T(n) \geq \log n$ and $P(n) \leq 2^{T(n)}$, $PRAM\text{-}TIME(T(n)) \subseteq RAM[\uparrow,\downarrow]\text{-}TIME(T(n) \log P(n)T(n))$.

Proof. By Lemma 9.2.2, BC', the family of bounded fan-in circuits that simulates a PRAM, is VM-uniform. By Theorem 9.1, a RAM[↑,↓] can simulate BC' in time $O(T(n) \log P(n)T(n))$. □

Lemma 9.3.1. Let $UC = \{UC_1, UC_2, ...\}$ be the family of unbounded fan-in circuits described in Lemma 6.2.2 that simulates a uniform PRAM[↑,↓]. UC is VM-uniform.

Proof. By Lemma 9.2.1, C is VM-uniform. UC has the same form as C, except in the blocks labeled [update common], handling updates to common memory. We reduce the inputs to the gates in this block because of restrictions on the processors that may simultaneously write a cell. It is easy to compute the processors that may simultaneously write a cell, so C is also VM-uniform. □

Lemma 9.3.2. Let $BC = \{BC_1, BC_2, ...\}$ be the family of bounded fan-in circuits described in Lemma 6.2.3 that simulates a uniform PRAM[↑,↓]. BC is VM-uniform.

Proof. Fix an input size n. BC_n is constructed from UC_n by replacing each gate with fan-in f by a tree of gates of depth $\log f$. A gate name in BC_n is the concatenation of the unbounded fan-in gate name in UC_n and the name of the gate within the bounded fan-in tree that replaces the unbounded fan-in gate (Figure 9.3). We prove VM-uniformity by the same algorithm given in the proof of Lemma 9.2.1, with modifications to test slot E, the portion of the gate name giving the gate name within the tree of depth $\log f$. By this algorithm, we see that the family BC of bounded fan-in circuits is VM-uniform since, by Lemma 9.3.1, UC is VM-uniform. □

Figure 9.3. Gate name in BC_n

Theorem 9.3. For all $T(n) \geq \log n$, $PRAM[\uparrow,\downarrow]\text{-}TIME(T(n)) \subseteq RAM[\uparrow,\downarrow]\text{-}TIME(T^3(n))$.

Proof. By Lemma 6.2.3 and Lemma 9.3.2, for each n, every language recognized by a PRAM[$\uparrow,\downarrow$] in time $T(n)$ can be recognized by a VM-uniform, bounded fan-in circuit BC_n of depth $O(T^3(n))$ and size $O(T^8(n)4^{T^2(n)})$. By Theorem 9.1, there exists a RAM[$\uparrow,\downarrow$] running in time $O(T^3(n) + (\log(T^8(n)4^{T^2(n)}))(\text{loglog}(T^8(n)4^{T^2(n)}))) = O(T^3(n))$ that simulates BC_n. □

Corollary 9.3.1. $PRAM[\uparrow,\downarrow]\text{-}PTIME = RAM[\uparrow,\downarrow]\text{-}PTIME$.

Combining Theorem 9.2 with Theorem 6.1 ($PRAM[\uparrow,\downarrow]\text{-}TIME(T(n)) \subseteq PRAM\text{-}TIME(T^2(n))$) implies

$$PRAM[\uparrow,\downarrow]\text{-}TIME(T(n)) \subseteq RAM[\uparrow,\downarrow]\text{-}TIME(T^4(n)).$$

The simulation of Theorem 9.3 is more efficient.

9.4. Simulation of MRAM-Uniform Circuit by RAM[*]

In this section, we adapt the simulation of a VM-uniform circuit by a RAM[$\uparrow,\downarrow$] (Section 9.2) to the case of a simulation of an MRAM-uniform circuit by a RAM[*].

Theorem 9.4. Let $MC = \{MC_1, MC_2, \cdots\}$ be a family of MRAM-uniform, bounded fan-in circuits of size $Z(n)$ and depth $D(n)$ recognizing language L. There exists a RAM[*] R that recognizes L in time $O(D(n) + \log Z(n) \text{ loglog } Z(n))$.

Proof. Without loss of generality, assume that R has two memories: mem_1 and mem_2. R performs the simulation described in Section 9.2, using a precomputed table of shift values in mem_2. To perform a left shift, such as $temp' \leftarrow temp \uparrow j$, R performs $temp' \leftarrow temp * 2^j$. To perform a right shift by j bits, R shifts all other values in mem_1 left by j bits, then notes that

the rightmost j bits of all registers are to be ignored (Hartmanis and Simon, 1974). This takes constant time because, by reusing registers, R uses only a constant number of registers in mem_1. In $O(\log Z(n))$ time, R computes the values $2^{Z(n)}$ and $2^{Z^2(n)}$, since $Z(n)$ and $Z^2(n)$ are the basic shift distances. In the course of the computation, R will perform shifts by $Z(n) \cdot 2^i$, $0 \le i \le \log Z(n)$, for each value of i. R computes the necessary shift value on each iteration from the previous value.

Thus, the simulation by R takes the same amount of time as the simulation described in Section 9.2: $O(D(n) + \log Z(n) \operatorname{loglog} Z(n))$. □

9.5. Simulation of PRAM[*] by RAM[*]

In this section, we simulate a PRAM[*] by an MRAM-uniform, bounded fan-in circuit family, then simulate this circuit family by a RAM[*]. We also simulate a basic PRAM by a RAM[*].

Lemma 9.5.1. Let $C = \{C_1, C_2, ...\}$ be the family of unbounded fan-in circuits described by Stockmeyer and Vishkin (1984) that simulates a uniform PRAM (Theorem 2.1). C is MRAM-uniform.

Proof. The lemma follows by the proof of Lemma 9.2.1. □

Let $PC = \{PC_1, PC_2, ...\}$ be the family of bounded fan-in circuits that simulates the family C of unbounded fan-in circuits described by Stockmeyer and Vishkin (1984). For a fixed input size n, the depth of PC_n is $O(T(n) \log P(n)T(n))$ and the size is $O(P(n)T(n)[T(n)(n+T(n)) + (n-T(n))^3 + (n+T(n))(n+P(n)T(n))])$.

Lemma 9.5.2. PC is MRAM-uniform.

Proof. By Lemma 9.5.1, C is MRAM-uniform. By the proof given for Lemma 9.3.2, PC is MRAM-uniform. □

Theorem 9.5. For all $T(n) \geq \log n$ and $P(n) \geq 2^{T(n)}$, $PRAM\text{-}TIME(T(n)) \subseteq RAM[*]\text{-}TIME(T(n) \log P(n) T(n))$.

Proof. By Lemma 9.5.2, PC, the family of bounded fan-in circuits that simulates a PRAM, is MRAM-uniform. By Theorem 9.4, a RAM[*] can simulate PC in time $O(T(n) \log P(n) T(n))$. □

Let BC denote the family of bounded fan-in circuits described in the proof of Lemma 4.2.2 that simulates a PRAM[*] in depth $O(T^2(n))$ and size $O(n^2 T^2(n)\, 8^{T(n)} \log T(n))$. We construct a family of bounded fan-in circuits BC' from BC. Fix an input size n. The circuit BC_n' is exactly the same as the circuit BC_n except that BC_n' uses a different multiplication block for reasons of MRAM-uniformity. Insert carry-save multiplication blocks in BC_n'. Each block has depth $O(T(n))$ and size $O(n^2 4^{T(n)})$. Thus, BC_n' has depth $O(T^2(n))$ and size $O(n^2 T^2(n)\, 8^{T(n)} \log T(n))$.

Lemma 9.6.1. For each n, every language recognized by a PRAM[*] R in time $T(n)$ with $P(n)$ processors can be recognized by the bounded fan-in circuit BC_n' of depth $O(T^2(n))$ and size $O(n^2 T^2(n)\, 8^{T(n)} \log T(n))$.

Proof. The proof is similar to that given for Lemma 4.2.2. □

Lemma 9.6.2. BC' is MRAM-uniform.

Proof. By Lemma 9.5.1, C is MRAM-uniform. By a proof like that given for Lemma 9.3.2, BC' is MRAM-uniform. □

Lemma 9.6.3. For each n, every language recognized by a PRAM[*] Y in time $T(n)$ with $P(n)$ processors can be recognized by a MRAM-uniform, bounded fan-in circuit BC_n' of depth $O(T^2(n))$ and size $O(T^{10}(n)\,P^4(n)\,16^{T(n)})$.

Proof. Fix an input length n. By Lemma 9.6.2, a bounded fan-in, MRAM-uniform circuit BC_n' of depth $O(T^2(n))$ and size $O(T^{10}(n)\,P^4(n)\,16^{T(n)})$ can simulate Y. □

Theorem 9.6. For all $T(n) \geq \log n$, $PRAM[*]\text{–}TIME(T(n)) \subseteq RAM[*]\text{–}TIME(T^2(n))$.

Proof. By Lemma 9.6.3, a PRAM[*] running in time $T(n)$ with $P(n)$ processors can be simulated by a bounded fan-in, MRAM-uniform circuit BC_n' of depth $O(T^2(n))$ and size $O(T^{10}(n)\,P^4(n)\,16^{T(n)})$. By Theorem 9.4, a RAM[*] can simulate BC_n' in time $O(T^2(n))$. □

9.6. Simulation of PRAM[*,÷] by RAM[*,÷]

In this section, we simulate a PRAM[*,÷] by a RAM[*,÷].

Theorem 9.7. For all $T(n) \geq \log n$, $PRAM[*,÷]\text{–}TIME(T(n)) \subseteq RAM[*,÷]\text{–}TIME(T^3(n))$.

Proof. By the proof of Theorem 5.1, a PRAM Z can simulate Y in time $O(T^2(n))$ with $O(P^2(n)\,T^2(n)\log T(n)\,n^2\,4^{T(n)})$ processors. By Theorem 9.5, a RAM[*], hence a RAM[*,÷], can simulate Z in time $O(T^3(n))$. □

Note that the family DC of bounded fan-in circuits in Lemma 5.2.1 simulates the PRAM[*,÷] in depth $O(T^2(n)\log T(n))$. We expect to show that DC is MRAM-uniform by proving that the division circuit of Shankar and Ramachandran (1987) is MRAM-uniform. This would lead to a $O(T^2(n)\log T(n))$ time simulation of a PRAM[*,÷] by a RAM[*,÷]. We are currently working on this problem.

As we noted in Chapter 5, a time-bounded RAM[+] is much weaker than a time-bounded PRAM[+]. Therefore, a simulation of a PRAM[+] by a RAM[+] would be highly inefficient.

Chapter 10. Alternatives

The PRAM is a flexible model, and many researchers have varied particular aspects of the model. Variations have arisen in whether to allow concurrent writes and, if so, in the rules governing concurrent writes, in whether to allow concurrent reads, in the amount of local memory allotted to each processor, in the use of shared memory, and in the mechanism for processor activation. Indeed, the focus of this thesis is relating various instruction sets in the PRAM model.

In this chapter, we discuss some of these variations and study their effects on our results.

• *Write conflict resolution*

We allow our PRAM concurrent read and concurrent write (CRCW) ability, resolving write conflicts by giving priority to the lowest numbered processor attempting to write. Fich *et al.* (1985) called this model the PRIORITY model. A number of other conflict resolution schemes exist: COMMON, in which all processors attempting to write must write the same value; ARBITRARY, in which an arbitrary processor succeeds in its write attempt; COLLISION, in which a special collision symbol appears in a cell if two or more processors attempt to write that cell simultaneously; and TOLERANT, in which the contents of a cell do not change in the event of a write conflict. PRIORITY is the strongest scheme. For a discussion of detailed relationships among these models, see Kucera (1982), Fich *et al.* (1985), Li and Yesha (1986), Fich *et al.* (1987), Grolmusz and Ragde (1987), Fich *et al.* (1988a), and Fich *et al.* (1988b).

• *CRCW vs. EREW*

We may restrict the model by disallowing concurrent writes, giving a concurrent read, exclusive write (CREW) version, or we may further restrict the model by disallowing concurrent reads, giving an exclusive read, exclusive write (EREW) version. For relationships among these restrictions, see Eckstein (1979), Vishkin (1983a), Snir (1985), Cook *et al.* (1986), Reischuk (1987), and Parberry and Yuan (1987).

In the preceding chapters, we presented relations among CRCW PRAMs with various instruction sets. We now prove similar results for EREW PRAMs, but with slightly higher time bounds, by converting the simulations by bounded fan-in circuits to simulations by EREW PRAMs.

Lemma 10.1.1. Let B $\{B_1, B_2, \cdots\}$ be a log-space uniform family of bounded fan-in circuits of depth $D(n) \leq \log n$ that accepts language L. There exists an EREW PRAM EP that runs in time $O(D(n))$ that accepts language L.

Proof. The proof is given in Karp and Ramachandran (1988). □

First, we simulate an EREW PRAM[*].

Theorem 10.1. $EREW\text{-}PRAM[*]\text{-}TIME(T(n)) \subseteq EREW\text{-}PRAM\text{-}TIME(T^2(n))$.

Proof. The theorem is true by Lemmas 4.2.2 and 10.1.1. □

Next, we simulate an EREW PRAM[*,+] and an EREW PRAM[+].

Theorem 10.2. $EREW\text{-}PRAM[*, +]\text{-}TIME(T(n)) \subseteq$
$EREW\text{-}PRAM\text{-}TIME(T^2(n) \log T(n))$.

Proof. The theorem is true by Lemmas 5.2.2 and 10.1.1. □

Lemma 10.3.1. An EREW PRAM can compute the quotient of two x bit operands in $O(\log x \log\log x)$ time.

Proof. The lemma is true by Lemmas 5.2.1 and 10.1.1. □

Lemma 10.3.2 states the key to converting CRCW simulations into EREW simulations.

Lemma 10.3.2. Let CR be a CRCW PRAM running in $T(n)$ time with $P(n)$ processors, and at most $Q(n)$ processors simultaneously read or write a single cell. An EREW PRAM ER can simulate CR in $O(T(n) \log Q(n))$ time.

Proof. A concurrent read or write by $Q(n)$ processors of CR takes $O(\log Q(n))$ time on ER by the method described by Vishkin (1983a), when we replace Batcher's sort with the faster parallel merge sort of Cole (1986). □

We must modify the Associative Memory Lemma to apply to EREW PRAMs.

Lemma 10.3.3. (EREW Associative Memory Lemma) Let $op \subseteq \{*, \uparrow, \downarrow\}$. For all $T(n)$ and $P(n)$, every language recognized with $P(n)$ processors in time $T(n)$ by an EREW PRAM[op] ER can be recognized in time $O(T(n) \log (P(n)T(n)))$ by an EREW PRAM[op] ER' that accesses only cells with addresses in 0, ..., $O(P(n)T(n))$.

Proof. At most $P(n)T(n)$ processors simultaneously read or write a single cell in the simulation presented in Chapter 3 in the proof of the Associative Memory Lemma. By Lemma 10.3.2, each step of this simulation can be simulated by ER' in $O(\log (P(n)T(n)))$ steps. □

Theorem 10.3. $EREW\text{-}PRAM[\div]\text{-}TIME(T(n)) \subseteq EREW\text{-}PRAM\text{-}TIME(T^2(n))$.

Proof. An EREW PRAM[÷] can generate numbers only up to $n + T(n)$ bits long; hence, by Lemma 10.3.1, an EREW PRAM takes $O(\log^2(n+T(n)))$ time to compute the quotient of

two such numbers. The EREW PRAM simulates the EREW PRAM[+] through Lemma 10.3.3, and the memory accesses of Lemma 10.3.3 dominate the computation time of each step. Hence, by Theorem 5.3 and Lemma 10.3.3, an EREW PRAM can simulate an EREW PRAM[+] running in time $T(n)$ in $O(T^2(n))$ steps. □

Now we simulate an EREW PRAM[↑,↓].

Theorem 10.4. *EREW-PRAM*[↑,↓]*-TIME*$(T(n)) \subseteq$ *EREW-PRAM-TIME*$(T^3(n))$.

Proof. The theorem is true by Lemmas 6.2.3 and 10.1.1. □

Finally, we simulate EREW PRAMs with probabilistic choice.

Theorem 10.5. Let $op \in \{\lambda, \{*\}, \{*,+\}, \{\uparrow,\downarrow\}, \{*,\uparrow,\downarrow\}\}$. Let R be a *prob*-RAM[op] with time bound $T(n)$ that makes $R(n)$ random choices. There is a deterministic EREW PRAM[op] *ED* that simulates R in $O(T(n))$ time with $2^{R(n)}$ processors.

Proof. The theorem follows exactly by the proof of Theorem 8.4. □

As in Chapter 8, we extend the simulation of a sequential machine to a simulation of a parallel machine. Again, we will need two proofs: one for PRAMs with enhanced instruction sets and one for basic PRAMs.

Theorem 10.6. Let $op \in \{\{*\}, \{*,+\}, \{\uparrow,\downarrow\}, \{*,\uparrow,\downarrow\}\}$. Let *EP* be an EREW *prob*-PRAM[op] with time bound $T(n)$, processor bound $P(n)$, and memory bound $S(n)$ that makes $R(n)$ random choices. There is a deterministic EREW PRAM[op] *ED* that simulates *EP* in time $O(R(n) + T(n) \log P(n))$ with $P(n)2^{R(n)}$ processors.

Proof. The proof follows the proof of Theorem 8.5, with a modification made for the exclusive read and exclusive write restrictions. The modification is in reading ζ_m. Up to $P(n)$ processors may wish to read ζ_m at each step, taking $O(\log P(n))$ time by Lemma

10.3.2. This, however, is the same amount of time required for sorting the write requests, so the time per step remains $O(\log P(n))$. Thus, the overall time for *ED* to simulate *EP* is $O(R(n) + T(n) \log P(n))$ steps. □

Theorem 10.7. Let *EP* be an EREW *prob*-PRAM with time bound $T(n)$, processor bound $P(n)$, and memory bound $S(n)$ that makes $R(n)$ random choices. There is a deterministic EREW PRAM *ED* that simulates *EP* in time $O(R(n) + T(n) \log P(n))$ with $P(n)2^{R(n)}$ processors.

Proof. The proof is that given for Theorem 10.6, with the exceptions listed in the proof of Theorem 8.6. □

We also present an EREW version of the Markov chain proof.

Lemma 10.8.1. Let *A* and *B* be $z \times z$ integer matrices stored one element per cell in the shared memory of an EREW PRAM[*] *E*. *E* can compute their product $C = AB$ in $O(\log z)$ time.

Proof. In $O(\log z)$ steps, *E* activates z^3 processors, assigning z processors to each element of matrix *C*. For each element $C(g, h)$ and all $1 \leq i \leq z$, the ith of its z processors computes $A(g, i) * B(i, h)$. Since each element of *A* and each element of *B* are read by z processors, *E* takes $O(\log z)$ time to read the elements. Next, also in $O(\log z)$ time, the processors assigned to each element of *C* add their products, writing the sum into the cell allocated to that element. □

As a preliminary step, we must describe a version of the Associative Memory Lemma for EREW *prob*-PRAMs.

Lemma 10.8.2. (Associative Memory Lemma for EREW *prob*-PRAMs) Let $op \subseteq \{*, +, \uparrow, \downarrow\}$. For all $T(n)$, $P(n)$, and $S(n)$, every language recognized in time $T(n)$ with $P(n)$

processors using at most $S(n)$ cells by an EREW *prob*-PRAM[*op*] EP can be recognized in time $O(T(n) \log (P(n)T(n)))$ by an EREW *prob*-PRAM[*op*] EP' that accesses only cells with addresses in 0, ..., $O(S(n))$.

Proof. By Lemma 8.7.1, a CRCW *prob*-PRAM[*op*] CP' running in time $O(T(n))$ with $O(P^2(n)T(n))$ processors recognizes every language recognized by EP. By Lemma 9.1.1, an EREW *prob*-PRAM[*op*] EP' simulates each step of CP' in time $O(\log(P(n)T(n)))$. □

Theorem 10.8. Let EP be an EREW *prob*-PRAM[*,÷] with time bound $T(n)$, processor bound $P(n)$, memory bound $S(n)$, integer bound $I(n)$, and program length k. Then there is a deterministic EREW PRAM[*,÷] ED that simulates EP in time $O((P(n) + \log I(n)) \cdot S(n) \cdot \log(T(n)))$ with $O((k^{P(n)} I(n))^{3S(n)})$ processors.

Proof (sketch). Let EP' simulate EP according to Lemma 10.8.2. Then EP' has time bound $O(T(n) \log (P(n)T(n)))$, processor bound $O(P(n)S(n))$, memory bound $S(n)$, and integer bound $I(n)$. By Lemma 10.8.1, an EREW PRAM[*,÷] can simulate ER' according to the simulation described in the proof of Theorem 8.7 in the same time. Since $P(n) \leq 2^{T(n)}$, $O(\log (T(n) \log (P(n)T(n)))) = O(\log T(n))$. □

• *Input convention*

PRAM definitions sometimes differ in the input convention. The input in our model is a single integer n bits long in $c(0)$. We have the special instruction $r(i) \leftarrow BIT(j)$, which places the $rcon(j)$th bit of the input in $r(i)$. Two other input styles are used: (1) the input consists of n bits, one each in $c(0)$, $c(1)$, ..., $c(n-1)$; (2) the input consists of r integers, one each in $c(0)$, $c(1)$, ..., $c(r-1)$, and the sum of the lengths of these integers is n. In $O(\log n)$ time, our PRAM can convert its input to either of the other styles by activating one processor for each bit position, then using the *BIT* instruction to read individual bits of the input. Note

that without the *BIT* instruction, the conversion would take $O(n)$ time on the basic PRAM because the PRAM must build bit masks n bits long to read individual bits of the input. Similarly, the basic PRAM takes $O(n)$ time to convert from either of the other styles to our input style. A PRAM[*] or PRAM[$\uparrow$,$\downarrow$] can convert the input from one style to another in $O(\log n)$ time.

• *Local memory*

We allow each processor infinite local memory. This definition was convenient, but not necessary, since a PRAM in which each processor has a constant number of local registers can simulate our PRAM with only a constant factor increase in time. We present two theorems to establish this fact, one for basic PRAMs and one for PRAMs with enhanced instruction sets.

Theorem 10.9. Let Z be a PRAM running in $T(n)$ time with $P(n)$ processors in which each processor has infinite local memory. A PRAM R in which each processor has only 4 local registers can simulate Z in $O(T(n))$ time with $P(n)$ processors.

Proof. We allow R three separate shared memories: mem_1, mem_2, and mem_3. R uses mem_1 to simulate the shared memory of Z, mem_2 to simulate the local memories of Z, and mem_3 to store an address table. In mem_2, R sets aside a block of $O(2^{T(n)})$ cells for each processor of R to use as the local memory of the corresponding processor of Z. For simplicity, assume the size of each block is $2^{T(n)}$. R will access $c_2(m\,2^{T(n)} + k)$ for every access to $r_m(k)$ of Z.

R activates $P(n)$ processors in $O(\log P(n))$ time. These processors make an address table in mem_3, storing $m\,2^{T(n)}$ in cell m, for $1 \le m \le P(n)$. This takes $O(T(n) + \log P(n))$ time. R uses this address table to speed access to the blocks of cells in mem_2.

Let P_m be the processor of R that corresponds to processor P_g of Z. When P_m is activated, it computes g and writes g in $r_m(0)$.

In a general step of Z, suppose processor P_g executes $r(i) \leftarrow r(j)\ O\ r(k)$. To simulate the read of the contents of $r_g(j)$, the corresponding processor P_m of R writes j in $r_m(2)$, copies $con_3(g)$ into $r_m(1)$, and adds j to $rcon_m(1)$. P_m then accesses $mem_2(g\,2^{T(n)}+j)$ indirectly through $r_m(1)$. P_m performs the same actions to read the contents of $r_g(k)$, except using registers 2 and 3 instead of 1 and 2. Now P_m performs $r(3) \leftarrow r(1)\ O\ r(2)$. R simulates the write in $r_g(i)$ just as described above.

R uses $O(T(n) + \log P(n)) = O(T(n))$ initialization time and constant time to simulate each step of Z. Thus, R simulates Z in $O(T(n))$ time with $P(n)$ processors. Each processor P_m of R uses only four registers $r_m(0), \cdots, r_m(3)$. □

Theorem 10.10. Let $op \subseteq \{*, \uparrow, \downarrow\}$. Let Z be a PRAM[op] running in $T(n)$ time with $P(n)$ processors in which each processor has infinite local memory. A PRAM[op] R in which each processor has only 4 local registers can simulate Z in $O(T(n))$ time with $P(n)$ processors.

Proof. We allow R two separate shared memories: mem_1 and mem_2. R uses mem_1 to simulate the shared memory of Z and mem_2 to simulate the local memories of Z. R will access $c_2(kP(n) + m)$ for $r_m(k)$ of Z. If $* \notin op$, then assume without loss of generality that $P(n)$ is a power of 2; this way R can perform shifts to perform the multiplications specified below.

R activates processors as specified by the program of Z, so processor P_m of R simulates processor P_m of Z. P_m of R stores $P(n)$ (or $\log P(n)$ if R does not have multiplication) in $r_m(1)$. In a general step of Z, suppose processor P_m executes $r(i) \leftarrow r(j)\ O\ r(k)$. To simulate the read of the contents of $r(j)$, the corresponding processor P_m of R writes j in

$r_m(2)$, multiplies $rcon_m(2)$ by $P(n)$, then adds m to the product. (By definition, P_m has m in $r_0(m)$.) P_m then reads $mem_2(jP(n)+m)$ indirectly through $r_m(2)$, writing $con_2(jP(n)+m)$ in $r_m(2)$. P_m performs the same actions to read the contents of $r(k)$, except using register 3 instead of 2. Now P_m performs $r(3) \leftarrow r(1)$ O $r(2)$. R simulates the write in $r(i)$ just as described above.

R uses constant time to simulate each step of Z. Thus, R simulates Z in $O(T(n))$ time with $P(n)$ processors. □

• *Operations in shared memory*

We restricted the instruction set so that the only operations permitted on shared memory are indirect reads and writes. Again, this definition was convenient, but not necessary, since such a PRAM can simulate a PRAM allowing all operations in shared memory with only a small constant factor increase in time.

Theorem 10.11. Let Z be a PRAM running in $T(n)$ time in which all instructions can be performed in either shared or local memory. A PRAM R allowing only indirect reads and writes to shared memory can simulate Z in $O(T(n))$ time.

Proof. For all m, processor P_m of R simulates processor P_m of Z. At time t, suppose P_m of Z executes $c(i) \leftarrow c(j)$ O $c(k)$. Then P_m of R copies $con(j)$ and $con(k)$ into its local memory, performs O, then writes $con(j)$ O $con(k)$ in $c(i)$. Thus, R simulates each step of Z in constant time. □

• *Processor activation*

A common method of processor activation is to assume that all $P(n)$ processors are initially active. Bounds for arbitrary $P(n)$ with this method of processor activation can be

derived from our simulations because we presented all simulations for arbitrary $P(n)$, then fixed $P(n) \leq 2^{T(n)}$ in deriving the final bounds.

The final area of alternative definitions is the *FORK* operation. Recall that we defined the instruction *FORK label* 1, *label* 2 as executed by P_g to cause P_g to halt and activate P_{2g} and P_{2g+1}, setting their program counters to *label* 1 and *label* 2, respectively. Fortune and Wyllie (1978) defined *FORK label* as executed by P_g to activate the lowest numbered inactive processor P_m, clear the local memory of P_m, copy the contents of the accumulator of P_g in the accumulator of P_m, and set the program counter of P_m to *label*. Let us call this operation *FW–FORK*. We now present a lemma to establish that our simulations all work with the same bounds with *FW–FORK* in the place of *FORK*.

The main difference between *FORK* and *FW–FORK* is in the processor number of the activated processor(s). In our simulation, the processor number is important in establishing the relationship between a primary and its secondary processors. Recall the Activation Lemma (Chapter 3): for a primary processor P_g with σ secondary processors, the secondary processors are numbered $k + g$, for $k = 1,...,\sigma$. Further, each secondary processor P_{g+k} computes k in order to assign itself to an item indexed by k in the computation. With *FW–FORK*, the relationship between processor numbers of primary and secondary processors is different: with π primary processors and σ secondary processors, the secondary processors belonging to primary processor P_g are numbered $k\pi + g$, for all $k = 1, ..., \sigma$. Once again, each secondary processor $P_{k\pi+g}$ must compute k in order to assign itself to an item indexed by k in the computation. These secondary processors activated by *FW–FORK* cannot determine k as easily as the processors activated by *FORK*, especially in a PRAM[*] without division. The next lemma describes a method by which the processors quickly determine k.

Ordering Lemma. Let π and g be fixed positive integers, $0 \le g \le \pi-1$, and let σ be another integer. Let Γ denote the set of processors $\{P_m \mid m = k\pi + g,\ 1 \le k \le \sigma\}$. Each processor P_m in Γ can determine k in $O(\log \sigma)$ time.

Proof. Assume π, σ, and g are known. In time $O(\log \sigma)$, one processor builds Table 1 in mem_1 such that location h contains the value 2^h, $0 \le h \le \lceil \log \sigma \rceil$. Also in time $O(\log \sigma)$, another processor builds Table 2 in mem_2 such that location h contains the value $\pi 2^h$, $0 \le h \le \lceil \log \sigma \rceil$. In the following, the values 2^h and $\pi 2^h$ are read from Table 1 and Table 2, respectively. Each processor P_m, $m = k\pi + g$, determines k as follows.

1. $\alpha := 1$, $\beta := 1$. (α, β are indices into the tables.)
2. Compare $\pi 2^\alpha$ with m. If $\pi 2^\alpha > m$, then $k := 1$.
3. $\alpha := \alpha + 1$.
4. Compare $\pi 2^\alpha$ with m. If $\pi 2^\alpha \le m$, then go to Step 3. If $\pi 2^\alpha > m$, then $\alpha/2 \le k < \alpha$. (P_m will determine the value of k within this range by binary search.)
5. $lower := \pi 2^{\alpha-1}$, $upper := \pi 2^\alpha$, and $k.bound := 2^{\alpha-1}$.
6. $\beta := \beta + 1$.
7. $middle := lower + \pi 2^{\alpha-\beta}$, $k.bound := k.bound + 2^{\alpha-\beta}$.
8. If $middle < m - g$, then $lower := middle$; go to Step 6.

   If $middle = m - g$, then $k := k.bound$. Done.

   If $middle > m - g$, then $upper := middle$ and $k.bound := k.bound - 2^{\alpha-\beta}$; go to Step 6.

P_m performs each step in the above algorithm in constant time. A processor may iterate Steps 6-8 or Steps 3 and 4 up to $O(\log \sigma)$ times. The processors build Tables 1 and 2 in

$O(\log \sigma)$ time. Thus, each processor in Γ can determine k in $O(\log \sigma)$ time. Note that the algorithm uses only addition and subtraction. □

Observe that $O(\log \sigma)$ is the same as the time required to activate σ processors, so the Ordering Lemma implies no more than a constant factor increase in time in a simulation if *FW –FORK* replaces *FORK*.

Chapter 11. Summary and Open Problems

11.1. Summary

In this thesis, we compared the computational power of time bounded Parallel Random Access Machines (PRAMs) with different instruction sets. We proved that polynomial time on PRAM[*]s or on PRAM[*,÷]s or on PRAM[↑,↓]s is equivalent to polynomial space on a Turing machine (*PSPACE*). In particular, we showed the following bounds. Let each simulated machine run for $T(n)$ steps on inputs of length n; let T denote $T(n)$ in the table below. The simulating machines are basic PRAM, Turing machine, RAM with the same instruction set, basic EREW PRAM, and uniform family of bounded fan-in circuits. The bounds for the simulating machine are expressed in time, space, or depth, as shown in parentheses by the machine type. The notation EREW means that the simulating machine is an EREW PRAM, and the simulated machine is an EREW PRAM[*op*].

Table 11.1. Summary of results

Simulating machine	Simulated machine			
	PRAM[*]	PRAM[*,÷]	PRAM[÷]	PRAM[↑,↓]
PRAM (time)	$T^2 / \log T$	T^2	$T \log(n+T)$	T^2
TM (space)	T^2	$T^2 \log T$	T^2	T^3
RAM[*op*] (time)	T^2	T^3	...	T^3
EREW (time)	T^2	$T^2 \log T$	T^2	T^3
circuit (depth)	T^2	$T^2 \log T$	T^2	T^3

As noted in Section 9.6, the simulation of a PRAM[÷] by a RAM[÷] is highly inefficient.

Further, we proved that *PRAM*[*, ↑,↓]–*PTIME* is contained between *NEXPTIME* and *EXPSPACE*. This is notable because polynomial time on a PRAM with either multiplication

or shifts alone is equivalent to *PSPACE*. Recall the Parallel Computation Thesis: polynomial time on a reasonable model of parallel computation is equivalent to polynomial space on a sequential model of computation. Our result of PRAM[*,↑,↓]s does not contradict the Parallel Computation Thesis because the numbers generated by multiplication and shift together are too long and complex to be "reasonable."

We also presented simulations of probabilistic PRAMs by deterministic PRAMs, using parallelism to replace randomness.

11.2. Open Problems

1. As noted in Chapter 1, if we could reduce the number of processors used by the simulation of a PRAM[*] or PRAM[*,÷] or PRAM[↑,↓] by a PRAM from an exponential number to a polynomial number, then *NC* would be the languages accepted by PRAM[*]s, PRAM[*,÷]s, or PRAM[↑,↓]s, respectively, in polylog time with a polynomial number of processors. Can the number of processors used by the PRAM in simulating the PRAM[*] be reduced to a polynomial in $P(n)T(n)$?

2. We showed $NEXPTIME \subseteq PRAM[*, \uparrow]\text{-}PTIME \subseteq EXPSPACE$ (Corollaries 7.1.1 and 7.2.1). Does $PRAM[*, \uparrow]\text{-}PTIME = EXPSPACE$?

3. What is the relationship between RAM[*,↑]s and PRAM[*,↑]s? Is $NEXPTIME \subseteq RAM[*, \uparrow]\text{-}PTIME$?

4. Can a log-space uniform, fan-in 2 $O(\log n)$ depth circuit perform division? Beame *et al.* (1986) developed a poly-time uniform division circuit. We could improve Theorems 5.1, 5.2, and 5.3 with a log-space uniform, $O(\log n)$ depth division circuit.

5. Can the $\log T(n)$ factor in $PRAM[*,+]\text{–}TIME(T(n)) \subseteq DSPACE(T^2(n) \log T(n))$ (Theorem 5.2) be removed by some other method?

6. What are the corresponding lower bounds on any of these simulations? Are any of the bounds optimal?

7. As one of the first results of computational complexity theory, the linear speed-up theorem for Turing machines (Hartmanis and Stearns, 1965) states that for every multitape Turing machine of time complexity $T(n) \gg n$ and every constant $c > 0$, there is a multitape Turing machine that accepts the same language in time $cT(n)$. The linear speed-up property of Turing machines justifies the widespread use of order-of-magnitude analyses of algorithms. Do PRAMs also enjoy the linear speed-up property?

Appendix A: Procedure *BOOL*

These appendices are written in guarded command style in order to more clearly show parallel cases. The general form of a conditional command is

if $B_1 \rightarrow S_1$
[] $B_2 \rightarrow S_2$
...
[] $B_n \rightarrow S_n$
fi

where B_i is a Boolean expression and S_i is a command or sequence of commands. For the deterministic case, exactly one guard B_i is true. Similarly, the form of a DO loop is

do $B \rightarrow S$
od

The loop is repeated until guard B is false. **skip** does nothing.

% The procedure *BOOL* (*j*, *k*, *i*, *v*) takes as input $B_i(g)$, containing the list of
% subtrees formed by merging the first levels of $E(rcon_g(j))$ and $E(rcon_g(k))$.
% *BOOL* returns the list with each subtree labeled as interesting or boring.
% For each node α, where α is the root of a subtree in the list, assume *proc* (α) is
% a secondary processor belonging to primary processor P_m,
% which corresponds to processor P_g of S'.
% For each α, *proc* (α) executes the steps specified below.
% The variables *j_int*, *k_int*, *nextj_int*, *nextk_int*, *right* (α), and *num* (α) are
% global variables.
% *j_int* tells if $rcon_g(j)$ is in an interval of 0's or 1's at the position specified by *val* (α).
% *k_int* is defined similarly.

if *right* (α) = 0 $\rightarrow$
% That is, if we want to know whether the least significant bit of $rcon_g(i)$ is 0 or 1.
if *j_int* = 1 **or** *k_int* = 1 $\rightarrow$ *result* := 1
[] *j_int* = 0 **and** *k_int* = 0 $\rightarrow$ *result* := 0
fi
[] *right* (α) $\neq$ 0 $\rightarrow$
% Otherwise, we want to know whether *val* (α) is the position of an interesting bit
% of $rcon_g(i)$.
if *val* (α) = *val* (*node* (*num* (α)+1)) $\rightarrow$ *result* := boring
% *result* = boring means that *val* (α) is not the location of an interesting bit in $rcon_g(i)$;
% *result* = interesting means that *val* (α) is the location of an interesting bit in $rcon_g(i)$.
% *proc* (α) tests whether or not *val* (α) = *val* (*node* (*num* (α)+1)) by calling *COMPARE*
[] *val* (α) $\neq$ *val* (*node* (*num* (α)+1)) $\rightarrow$
% *nextj_int* is *j_int* at *val* (*node* (*num* (α)+1)); *nextk_int* is defined similarly.
if *j_int* = 1 **or** *k_int* = 1 $\rightarrow$
if *nextj_int* = 1 **or** *nextk_int* = 1 $\rightarrow$ *result* := boring
[] *nextj_int* = 0 **and** *nextk_int* = 0 $\rightarrow$ *result* := interesting
fi
[] *j_int* = 0 **and** *k_int* = 0 $\rightarrow$

```
            if nextj_int = 1 or nextk_int = 1 → result := interesting
                [] nextj_int = 0 and nextk_int = 0 → result := boring
                fi
            fi
        fi
    fi
```

Appendix B: Procedure *ADD* (PRAM)

```
% The procedure ADD (j, ψ1, k, ψ2, i, ψ3) takes as input B_i(g), containing
% the list of subtrees formed by merging the first levels of E(rcon_g(j))
% and E(rcon_g(k)).
% ADD returns the list with each subtree labeled as interesting or boring,
% and the value of each interesting subtree specifies the location of an
% interesting bit in rcon_g(i) = rcon_g(j) + rcon_g(k).
% For each node α, where α is the root of a subtree in the list, assume proc (α) is
% a secondary processor belonging to primary processor P_m,
% which corresponds to processor P_g of S'.
% For each α, proc (α) executes the steps specified below.
% The variables i_int, j_int, k_int, nextj_int, nextk_int, carryout, pair_length,
% right (α), and num (α) are global variables.
% j_int tells if rcon_g(j) is in an interval of 0's or 1's at the position specified
% by val (α); k_int and i_int are defined similarly.
% pair_length tells if the interval-pair length of the interval-pair ending at
% position val (α) is one or more.
% carryin tells whether there is a carry into an interval-pair.


if right (α) = 0 →
% That is, if we want to know whether the least significant bit of rcon_g(i) is 0 or 1.
   if j_int = k_int → result := 0
   [] j_int ≠ k_int → result := 1
   fi
[] right (α) ≠ 0 →
% Otherwise, we want to know whether val (α) is the position of an interesting bit
%  of rcon_g(i).
   if val (α) = val (node (num (α)+1)) → result := boring
% result = boring means that val (α) is not the location of an interesting bit in rcon_g(i);
% result = interesting means that val (α) is the location of an interesting bit in rcon_g(i).
   [] val (α) ≠ val (node (num (α)+1)) →
% nextj_int is j_int at val (node (num (α)+1)); nextk_int is defined similarly.

      if (nextj_int = nextk_int = carryout ≠ i_int)
            or ((nextj_int ≠ nextk_int) ∧ (i_int = carryout)) →
         if pair_length = 1 → result := boring
         [] pair_length = more →
            E(rcon_g(i).ψ3.(right (α)+1)
               := ADD (i, ψ3.(right (α)+1), # 1, λ, i, ψ3.(right (α)+1))
            result := interesting
         fi
% right (α)+1 specifies which element of the merged list α is, counting from the right.

      [] nextj_int = nextk_int = i_int ≠ carryout →
         if pair_length = 1 → result := interesting
```

```
        [] pair_length = more →
                result := interesting
                E(rcon_g(i).ψ3.(right(α)+1½)
                    := ADD (i, ψ3.(right(α)+1), #1, λ, i, ψ3.(right(α)+1))
                result := interesting
% In this case, an interesting bit occurs in the sum at val(α) and val(α)+1.
% We say that we insert the second interesting bit into the merged list at
% location right(α)+1½.
% In fact, the merging algorithm leaves an empty slot between each pair of
% consecutive elements so that such interesting bits may be inserted.


        [] ((nextj_int ≠ nextk_int) ∧ (i_int ≠ carryout))
                or (nextj_int = nextk_int = i_int = carryout) → skip
        fi
    fi
fi
```

Appendix C: Procedure *COMPARE*

```
% Assume α has the form j, β has the form k.
% Assume rcon(j) and rcon(k) are positive.
% COMPARE recursively compares subtrees of I(rcon(j)).ψ1 and I(rcon(k)).ψ2
% from right to left.
% Let I(α') = I(rcon(j)); I(β') = I(rcon(k)).
% Return "greater" if val(I(α').ψ1) > val(I(β').ψ2),
%        "equal" if val(I(α').ψ1) = val(I(β').ψ2), or
%        "less" if val(I(α').ψ1) < val(I(β').ψ2) at time t.

Sα := SYMBOL(α, ψ1, t);
Sβ := SYMBOL(β, ψ2, t);
if Sα = subtree and Sβ ≠ subtree  →  result := greater
[] Sα ≠ subtree and Sβ = subtree  →  result := less
[] Sα = 1 and Sβ = 0  →  result := greater
[] Sα = 0 and Sβ = 1  →  result := less
[] Sα = 0 and Sβ = 0  →  result := equal
[] Sα = 1 and Sβ = 1  →  result := equal
[] Sα = subtree and Sβ = subtree  →
    A := SYMBOL(α, ψ1.1, t);
    B := SYMBOL(β, ψ2.1, t);
% A tells whether we are currently looking at a run of 0's or 1's in rcon(j);
% similarly for B. Their initial values tell if rcon(j) and rcon(k) start in a
% run of 0's or 1's
    if A > B  →  result := greater
    [] A = B  →  result := equal
    [] A < B  →  result := less
    fi;
    aψ := 2;  bψ := 2;
% aψ and bψ are pointers into their respective encodings
    do neither aψ nor bψ reaches beyond encoding  →
        X := COMPARE(α, ψ1.aψ, β, ψ2.bψ, t);

% If A = B, then we do not update result.
% If A ≠ B, then we update result to indicate which is greater.
% In both cases, we advance the pointers and update A and B as needed.

        if X = greater  →  bψ := bψ + 1;  B := ¬B
        [] X = equal  →  aψ := aψ + 1;  bψ := bψ + 1;  A := ¬A;  B := ¬B
        [] X = less  →  aψ := aψ + 1;  A := ¬A
        fi;
        if A = 0 and B = 1  →  result := less
        [] A = 1 and B = 0  →  result := greater
        fi
    od
```

```
% at this point, either aψ or bψ points past its encoding
    if only aψ points past encoding → result := less
% that is, |rcon(j)| < |rcon(k)|, so rcon(j) < rcon(k)
    [] only bψ points past encoding → result := greater

% if both point past encodings, then we must test which is greater
    [] both aψ and bψ point past their encodings →
        X := COMPARE(α.ψ1, aψ−1, β.ψ2, bψ−1, t);
        if X = greater or X = less → result := X
        [] X = equal → skip
        fi
% otherwise, result stays the same
    fi
fi
```

Recall that α and β can have five different forms. These are: j, $j.\phi$, $\#d$, $1+j.\phi$, and $1+\overline{j.\phi}$. The above algorithm considers only the first form. For α of the form $j.\phi$, call *COMPARE* (j, $\phi.\psi_1$, k, ψ_2, t). For α of the form $\#d$, call *CONVERT*($\#d$) to convert the constant d to the interesting bit encoding. For α of the form $1+j.\phi$, call *COMPARE* (*ADD* ($j.\phi$, # 1, ψ_1, t), λ, k, ψ_2, t). For α of the form $1+\overline{j.\phi}$, handle α the same as in the previous case, except interpret 0's as 1's and 1's as 0's.

Appendix D: Procedure *SYMBOL*, Boolean Case

```
% Assume γ has the form i.
% Suppose we have found that processor P_m executed instruction instr at time t−1
% that wrote r(i). instr was r(i)←r(j) ∨ r(k).
% If γ.ψ points to a leaf in I(γ) at time t, then return the symbol (0 or 1),
% otherwise, return an indication that γ.ψ points to a subtree.

if ψ = λ →
% λ represents the empty pointer
    j_run := SYMBOL(j, λ, t−1);
    k_run := SYMBOL(k, λ, t−1);
    if j_run = subtree or k_run = subtree → result := subtree
    [] j_run = 1 or k_run = 1 → result := 1
    [] j_run = 0 and k_run = 0 → result := 0
    fi
[] ψ ≠ λ →
    j_run := SYMBOL(j, 1, t−1);
    k_run := SYMBOL(k, 1, t−1);
% j_run tells if we are currently looking at a run of 0's or 1's in rcon(j),
% similarly for k_run
    if ψ = 1 →
% that is, if we want to know whether the least significant bit of γ is 0 or 1
        if j_run = 1 or k_run = 1 → result := 1
        [] j_run = 0 and k_run = 0 → result := 0
        fi
    [] ψ ≠ 1 →
% else we want to know the location of some interesting bit of γ
        jψ := 2; kψ := 2; γψ := 2;
% jψ, kψ, and γψ are pointers to show where we are currently
% looking in the respective encodings
        result := λ;
        do result = λ →
            oldj_run := j_run;
            oldk_run := k_run;
            runstop := COMPARE(j, jψ, k, kψ, t−1);
% runstop indicates which run of identical bits stops first
            if runstop = less → string := j
% string tells the string, j, k, or both, whose run of bits stops first
            [] runstop = equal → string := both
            [] runstop = greater → string := k
            fi;
            if string = j → j_run := ¬ j_run; jψ := jψ + 1
            [] string = both →
                j_run := ¬ j_run; k_run := ¬ k_run;
                jψ := jψ + 1; kψ := kψ + 1
            [] string = k → k_run := ¬ k_run; kψ := kψ + 1
```

```
            fi;
                if oldj_run = 0 and oldk_run = 0 →
                    if FIRST(ψ) = γψ →
                        if string = both or string = j →
                            result := SYMBOL(j.jψ, REST(ψ), t−1)
                        [] string = k → result := SYMBOL(k.kψ, REST(ψ), t−1)
                        fi
                    [] FIRST(ψ) ≠ γψ → γψ := γψ + 1
                    fi
                [] oldj_run = 1 or oldk_run = 1 →
                    if j_run = 0 and k_run = 0 →
% then an interesting bit occurs at the end of the 1's
                        if FIRST(ψ) = γψ →
                            if string = both or string = j
                            → result := SYMBOL(j.(jψ−1), REST(ψ), t−1)
                            [] string = k →
                               result := SYMBOL(k.(kψ−1), REST(ψ), t−1)
                            fi
                        [] FIRST(ψ) ≠ γψ → γψ := γψ + 1
                        fi
                    [] j_run = 1 or k_run = 1 → skip
                    fi
                fi
            od
        fi
    fi
```

Appendix E: Procedure *ADD* (TM)

```
% Assume α has the form j, β has the form k.
% γ ← α + β, so γ ← r(j) + r(k)
% Return result := I(γ).ψ.

if ψ = λ →
    j_run := SYMBOL(j, λ, t−1);
    k_run := SYMBOL(k, λ, t−1);
% j_run tells whether we are looking at a run of 0's or 1's
% in rcon(j); similarly for k_run and γ_run
    if j_run = 1 and k_run = 1 → result := subtree
    [] j_run = 0 or k_run = 0 → result := j_run ∨ k_run
    fi
[] ψ = 1 →
    j_run := SYMBOL(j, 1, t−1);
    k_run := SYMBOL(k, 1, t−1);
    if j_run = k_run → result := 0
    [] j_run ≠ k_run → result := 1
    fi
[] ψ ≠ λ and ψ ≠ 1 →
    j_run := SYMBOL(j, 1, t−1);
    k_run := SYMBOL(k, 1, t−1);
    if j_run = k_run → γ_run := 0
    [] j_run ≠ k_run → γ_run := 1
    fi;
    carryin := 0;  jψ := 2;  kψ := 2;  γψ := 2;
% carryin tells whether there is a carry into a run-pair;
% jψ, kψ, γψ are pointers into their respective encodings
    finished := false;
% finished is used as a flag for exiting the do statement
    do ¬ finished →
        right_pointer := COMPARE(j, jψ, k, kψ, t−1);
% tells which pointer is to the right, that is, end of current run-pair
        old_left := COMPARE(j, (jψ−1), k, (kψ−1), t−1);
% old_left indicates end of previous run-pair
        if (right_pointer = less or right_pointer = equal)
                and (old_left = less or old_left = equal) →
% right_pointer = less or equal means val(j.jψ) ≤ val(k.kψ)
            if COMPARE(j, jψ, 1+k.(kψ−1), t−1) = equal → pair_length := 1
% pair_length tells if the run-pair is of length one or more
            [] COMPARE(j, jψ, 1+k.(kψ−1), t−1) ≠ equal → pair_length := more
            fi
        [] (right_pointer = less or right_pointer = equal) and old_left = greater →
            if COMPARE(j, jψ, 1+j.(jψ−1), t−1) = equal → pair_length := 1
            [] COMPARE(j, jψ, 1+j.(jψ−1), t−1) ≠ equal → pair_length := more
            fi
```

```
    [] right_pointer = greater and (old_left = less or old_left = equal) →
            if COMPARE (k, kψ, 1+k. (kψ−1), t−1) = equal → pair_length := 1
            [] COMPARE (k, kψ, 1+k. (kψ−1), t−1) ≠ equal → pair_length := more
            fi
        [] right_pointer = greater and old_left = greater →
            if COMPARE (k, kψ, 1+j. (jψ−1), t−1) = equal → pair_length := 1
            [] COMPARE (k, kψ, 1+j. (jψ−1), t−1) ≠ equal → pair_length := more
            fi
        fi;

        if γψ = FIRST (ψ) →
% then this is the desired subtree
            if γψ = ψ → ξ := λ
% then this is the desired answer
            [] γψ ≠ ψ → ξ := REST (ψ)
% ξ is a pointer
            fi;
            finished := true;
        [] γψ ≠ FIRST (ψ) → skip
        fi

        if (j_run = k_run = carryin ≠ γ_run)
                or ((j_run ≠ k_run) ∧ (γ_run = carryin)) →
            if γψ = FIRST (ψ) →
                old_left := COMPARE (j, (jψ−1), k, (kψ−1), t−1);
                if old_left = greater or old_left = both →
                        result := SYMBOL (j, (jψ−1).ξ, t−1)
                [] old_left = less → result := SYMBOL (k, (kψ−1).ξ, t−1)
                fi
            [] γψ ≠ FIRST (ψ) → γψ := γψ + 1; γ_run := ¬ γ_run
            fi

        [] j_run = k_run ≠ γ_run = carryin →
            if pair_length = 1 →
                carryin := ¬ carryin;
                if γ_run = 1 → finished := false
% set finished to false in case it was earlier set true, no interesting bit occurs
% in this case
                [] γ_run = 0 → skip
                fi
            [] pair_length = more →
                if γψ = FIRST (ψ) →
                    old_left := COMPARE (j, (jψ−1), k, (kψ−1), t−1);
                    if old_left = greater or old_left = both
                      → result := ADD (j. (jψ−1), # 1, ξ, t−1)
                    [] old_left = less → result := ADD (k. (kψ−1), # 1. ξ, t−1)
                    fi
```

```
                [] γψ ≠ FIRST(ψ) → γψ := γψ + 1;  carryin := ¬ carryin;
                      γ_run := ¬ γ_run
                fi
            fi

        [] j_run = k_run = γ_run ≠ carryin →
            if pair_length = 1 →
                if γψ = FIRST(ψ) →
                    old_left := COMPARE(j, (jψ−1), k, (kψ−1), t−1);
                    if old_left = greater or old_left = both
                      → result := SYMBOL(j, (jψ−1).ξ, t−1)
                    [] old_left = less → result := SYMBOL(k, (kψ−1).ξ, t−1)
                    fi
                [] γψ ≠ FIRST(ψ) → γψ := γψ + 1;  carryin := ¬ carryin;
                  γ_run := ¬ γ_run
                fi
            [] pair_length = more →
                if γψ = FIRST(ψ) →
                    old_left := COMPARE(j, (jψ−1), k, (kψ−1), t−1);
                    if old_left = greater or old_left = both
                    → result := SYMBOL(j, (jψ−1).ξ, t−1)
                    [] old_left = less → result := SYMBOL(k, (kψ−1).ξ, t−1)
                    fi
                [] γψ ≠ FIRST(ψ) →
% pick up the isolated interesting bit
                    γψ := γψ + 1;
                    if γψ = FIRST(ψ) →
                        if γψ = ψ → ξ := λ
                        [] γψ ≠ ψ → ξ := REST(ψ)
                        fi;
                        finished := true;
% to get out of do loop
                        old_left := COMPARE(j, (jψ−1), k, (kψ−1), t−1);
                        if old_left = greater or old_left = both
                        → result := ADD(j.(jψ−1), #1, ξ, t−1)
                        [] old_left = less → result := ADD(k.(kψ−1), #1, ξ, t−1)
                        fi
                    [] γψ ≠ FIRST(ψ) → γψ := γψ + 1;  carryin := ¬ carryin
                    fi
                fi
            fi

        [] ((j_run ≠ k_run) ∧ (γ_run ≠ carryin))
            or (j_run = k_run = γ_run = carryin) → skip
        fi;

% now shift jψ and kψ as necessary to point to the next interesting bit in
```

```
% their respective encodings
        if right_pointer = less → jψ := jψ + 1;  j_run := ¬ j_run
% that is, if the run of identical bits in rcon(j) stops before the
% run of identical bits in rcon(k)
        [] right–pointer = equal →
            jψ := jψ + 1;  kψ := kψ + 1;
            j_run := ¬ j_run;    k_run := ¬ k_run
        [] right_pointer = greater → kψ := kψ + 1;  k_run := ¬ k_run
        fi;

% now handle the instance when at least one of jψ, kψ points beyond its encoding
        if ¬ finished →
            if jψ and kψ point beyond their encodings →
% if both point beyond their encodings, we must check if a carry out
% from the last run-pair causes one more interesting bit to occur in I(γ)
                if carryin = 0 →
                    result := beyond;  finished := true
% beyond means that I(γ) has fewer than FIRST(ψ)-1 interesting bits
                [] carryin = 1 →
% carryin = 1 and rcon(γ) has a 1, which is an interesting bit,
% in the location beyond the interesting bits of rcon(j) and rcon(k)
                    if γψ ≠ FIRST(ψ) → result := beyond;  finished := true
                    [] γψ = FIRST(ψ) →
% this is the bit we are looking for
                        old_left := COMPARE(j, (jψ–1), k, (kψ–1), t–1);
                        if old_left = greater or old_left = both
                        → result := ADD(j.(jψ–1), #1, ξ, t–1)
                        [] old_left = less → result := ADD(k.(kψ–1), #1, ξ, t–1)
                        fi
                    fi
                fi

            [] kψ does not point beyond its encoding →
                result := SYMBOL(k, FIRST(ψ) – γψ + kψ – 2).REST(ψ), t–1);
% FIRST(ψ) – γψ - 1 gives the remaining interesting bits to pass
% over in I(γ); k(ψ)– 1 gives the interesting bits in
% I(rcon(k)) already accounted for
                finished := true

            [] jψ does not point beyond its encoding →
                result := SYMBOL(j, FIRST(ψ) – γψ + jψ – 1).REST(ψ), t–1);
                finished := true
            fi
        [] finished → skip
        fi
    od
fi
```

References

H. Alt, T. Hagerup, K. Mehlhorn, and F. P. Preparata (1987), "Deterministic Simulation of Idealized Parallel Computers on More Realistic Ones," *SIAM J. Comput.*, vol. 16, no. 5, pp. 808-835.

P. B. Beame, S. A. Cook, and H. J. Hoover (1986), "Log Depth Circuits for Division and Related Problems," *SIAM J. Comput.*, vol. 15, no. 4, pp. 994-1003.

A. Borodin (1977), "On Relating Time and Space to Size and Depth," *SIAM J. Comput.*, vol. 6, pp. 733-744.

A. Borodin and J. E. Hopcroft (1985), "Routing, Merging, and Sorting on Parallel Models of Computation," *J. Comput. Syst. Sci.*, vol. 30, pp. 130-145.

A. K. Chandra, S. Fortune, and R. Lipton (1985), "Unbounded Fan-in Circuits and Associative Functions," *J Comput. Syst. Sci.*, vol. 30, pp. 222-234.

A. K. Chandra and L. J. Stockmeyer (1976), "Alternation," *Proc. 17th IEEE Symp. Foundations Comput. Sci.*, pp. 98-108.

A. K. Chandra, L. J. Stockmeyer, and U. Vishkin (1984), "Constant Depth Reducibility," *SIAM J. Comput.*, vol. 13, no. 2, pp. 423-439.

S. C. Chen (1983) "Large-Scale and High-Speed Multiprocessor System for Scientific Application--Cray X-MP Series," *Proc. NATO Adv. Res. High Speed Comput.*, pp. 117-126.

R. Cole (1986), "Parallel Merge Sort," *Proc. 27th IEEE Symp. Foundations Comput. Sci.*, pp. 511-516.

S. A. Cook (1980), "Towards a Complexity Theory of Synchronous Parallel Computation," Technical Report 141/80, Dept. of Computer Science, University of Toronto.

S. A. Cook (1985), "A Taxonomy of Problems with Fast Parallel Algorithms," *Inf. Control*, vol. 64, pp. 2-22.

S. Cook, C. Dwork, and R. Reischuk (1986), "Upper and Lower Time Bounds for Parallel Random Access Machines Without Simultaneous Writes," *SIAM J. Comput.*, vol. 15, no. 1, pp. 87-97.

S. A. Cook and R. A. Reckhow (1973), "Time Bounded Random Access Machines," *J. Comput. Syst. Sci.*, vol. 7, pp. 354-375.

P. W. Dymond and S. A. Cook (1980), "Hardware Complexity and Parallel Complexity," *Proc. 21st IEEE Symp. Foundations Comput. Sci.*, pp. 360-372.

P. W. Dymond and M. Tompa (1985), "Speedups of Deterministic Machines by Synchronous Parallel Machines," *J. Comput. Syst. Sci.*, vol. 30, pp. 149-161.

D. M. Eckstein (1979), "Simultaneous Memory Accesses," Technical Report TR-79-6, Computer Science Dept., Iowa State University.

F. E. Fich, F. Meyer auf der Heide, P. Ragde, and A. Wigderson (1985), "One, Two, Three ... Infinity: Lower Bounds for Parallel Computation," *Proc. 17th ACM Symp. Theory Comput.*, pp. 48-58.

F. E. Fich, F. Meyer auf der Heide, and A. Wigderson (1987), "Lower Bounds for Parallel Random-Access Machines with Unbounded Shared Memory," *Advances in Computing Research*, ed. F. P. Preparata, vol. 4, JAI Press, pp. 1-15.

F. E. Fich, P. Ragde, and A. Wigderson (1988a), "Simulations Among Concurrent-Write PRAMs," *Algorithmica*, vol. 3, pp. 43-51.

F. E. Fich, P. Ragde, and A. Wigderson (1988b), "Relations Between Concurrent-Write Models of Parallel Computation," *SIAM J. Comput.*, vol. 17, no. 3, pp. 606-627.

S. Fortune and J. Wyllie (1978), "Parallelism in Random Access Machines," *Proc. 10th ACM Symp. Theory Comput.*, pp. 114-118.

J. Gill (1977), "Computational Complexity of Probabilistic Turing Machines," *SIAM J. Comput.*, vol. 6, no. 4, pp. 675-695.

L. M. Goldschlager (1978), "A Unified Approach to Models of Synchronous Parallel Machines," *Proc. 10th ACM Symp. Theory Comput.*, pp. 89-94.

L. M. Goldschlager (1982), "A Universal Interconnection Pattern for Parallel Computers," *J. ACM*, vol. 29, no. 3, pp. 1073-1086.

A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir (1983), "The NYU Ultracomputer---Designing an MIMD Shared Memory Parallel Computer," *IEEE Trans. Comput.*, vol. C-32, no. 2, pp. 175-189.

V. Grolmusz and P. Ragde (1987), "Incomparability in Parallel Computation," *Proc. 28th IEEE Symp. Foundations Comput. Sci.*, pp. 89-98.

J. Hartmanis (1971), "Computational Complexity of Random Access Stored Program Machines," *Math. Syst. Theory*, vol. 5, no. 3, pp. 232-245.

J. Hartmanis and J. Simon (1974), "On the Power of Multiplication in Random Access Machines," *Proc. 15th Symp. Switching Automata Theory*, pp. 13-23.

J. Hartmanis and R. E. Stearns (1965), "On the Computational Complexity of Algorithms," *Trans. Amer. Math. Soc.*, vol. 117, pp. 285-306.

J. W. Hong (1986), *Computation: Computability, Similarity and Duality*, Wiley, New York.

J. E. Hopcroft, W. J. Paul, and L. G. Valiant (1975), "On Time Versus Space and Related Problems," *Proc. 16th IEEE Symp. Foundations Comput. Sci.*, pp. 57-64.

K. Hwang (1979), *Computer Arithmetic: Principles, Architecture, and Design*, Wiley, New York.

R. M. Karp and V. Ramachandran (1988), "A Survey of Parallel Algorithms for Shared-Memory Machines," Univ. of California at Berkeley Tech. Report No. UCB/CSD 88/408.

J. Katajainen, J. van Leeuwen, and M. Penttonen (1988), "Fast Simulation of Turing Machines by Random Access Machines," *SIAM J. Comput.*, vol. 17, no. 1, pp. 77-88.

L. Kucera (1982), "Parallel Computation and Conflicts in Memory Access," *Inf. Process. Lett.*, vol. 14, pp. 93-96.

D. J. Kuck (1986), "Parallel Computing Today and Cedar Approach," *Science*, pp. 967-974.

R. E. Ladner and M. J. Fischer (1980), "Parallel Prefix Computation," *J. ACM*, vol. 27, pp. 831-838.

K. de Leeuw, E. F. Moore, C. E. Shannon, and N. Shapiro (1956), "Computability by Probabilistic Machines," *Automata Studies, Annals of Mathematical Studies 34*, Princeton University Press, Princeton, NJ, pp. 183-212.

M. Li and Y. Yesha (1986), "New Lower Bounds for Parallel Computation," *Proc. 18th ACM Symp. Theory Comput.*, pp. 177-187.

M. Luby (1986), "A Simple Parallel Algorithm for the Maximal Independent Set Problem," *SIAM J. Comput.*, vol. 15, no. 4, pp. 1036-1053.

G. L. Miller and J. H. Reif (1985), "Parallel Tree Contraction and Its Application," *Proc. 26th IEEE Symp. Foundations Comput. Sci.*, pp. 478-489.

W. Morris, ed. (1980), *The American Heritage Dictionary of the English Language*, Houghton Mifflin, Boston.

I. Parberry (1986), "Parallel Speedup of Sequential Machines: A Defense of the Parallel Computation Thesis," *SIGACT News*, vol. 18, no. 1, pp. 54-67.

I. Parberry (1987), *Parallel Complexity Theory*, Wiley, New York.

I. Parberry and P. Y. Yuan (1987), "Improved Upper and Lower Time Bounds for Parallel Random Access Machines without Simultaneous Writes," Technical Report CS-87-29, Dept. of Computer Science, Pennsylvania State University.

G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss (1985), "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proc. Int. Conf. Par. Process.*, pp. 764-771.

V. Pratt and L. Stockmeyer (1976), "A Characterization of the Power of Vector Machines," *J. Comput. Syst. Sci.*, vol. 12, pp. 198-221.

F. P. Preparata and J. Vuillemin (1981), "The Cube-Connected Cycles: A Versatile Network for Parallel Computation," *Commun. ACM*, vol. 24, no. 5, pp. 300-309.

M. O. Rabin (1976), "Probabilistic Algorithms," *Algorithms and Complexity*, edited by J. Traub, Academic Press, New York.

S. Rajasekaran and J. H. Reif (1987), "Randomized Parallel Computation," *Lect. Notes Comput. Sci. 270* (Computation Theory and Logic), pp. 364-376.

A. G. Ranade (1987), "How to Emulate Shared Memory," *Proc. 28th IEEE Symp. Foundations Comput. Sci.*, pp. 185-194.

J. H. Reif (1984), "On Synchronous Parallel Computations with Independent Probabilistic Choice," *SIAM J. Comput.*, vol. 13, no. 1, pp. 46-56.

J. H. Reif (1986), "Logarithmic Depth Circuits for Algebraic Functions," *SIAM J. Comput.*, vol. 15, no. 1, pp. 231-242.

R. Reischuk (1987), "Simultaneous WRITES of Parallel Random Access Machines Do Not Help to Compute Simple Arithmetic Functions," *J. ACM*, vol. 34, no. 1, pp. 163-178.

J. M. Robson (1984), "Fast Probabilistic RAM Simulation of Single Tape Turing Machine Computations," *Inf. Control*, vol. 63, pp. 67-87.

W. L. Ruzzo (1981), "On Uniform Circuit Complexity," *J. Comput. Syst. Sci.*, vol. 22, pp. 365-383.

W. Ruzzo (1985), "The Equivalence of Restricted Parallel Random Access Machines and Hardware Modification Machines," preprint.

E. S. Santos (1969), "Probabilistic Turing Machines and Computability," *Proc. Amer. Math. Soc.*, vol. 22, pp. 704-710.

E. S. Santos (1971), "Computability by Probabilistic Turing Machines," *Trans. Amer. Math. Soc.*, vol. 159, pp. 165-184.

W. Savitch (1982), "Parallel Random Access Machines with Powerful Instruction Sets," *Math. Syst. Theory*, vol. 15, pp. 191-210.

W. Savitch and M. Stimson (1979), "Time Bounded Random Access Machines with Parallel Processing," *J. ACM*, vol. 26, no. 1, pp. 103-118.

A. Schönhage (1979), "On the Power of Random Access Machines," *Lect. Notes Comput. Sci. 71*, ("Automata, Languages, and Programming 6th Colloquium,"), pp. 520-529.

A. Schönhage (1980), "Storage Modification Machines," *SIAM J. Comput.*, vol. 9, no. 3, pp. 490-508.

A. Schönhage and V. Strassen (1971), "Schnelle Multiplikation grosser Zahlen," *Computing*, vol. 7, pp. 281-292.

J. T. Schwartz (1980), "Ultracomputers," *ACM Trans. Program. Lang. Syst.*, vol. 2, no. 4, pp. 484-521.

C. L. Seitz (1985), "The Cosmic Cube," *Commun. ACM*, vol. 28, no. 1, pp. 22-33.

N. Shankar and V. Ramachandran (1987), "Efficient Parallel Circuits and Algorithms for Division," Technical Report UILU-ENG-87-2235 (ACT-78), Coordinated Science Laboratory, University of Illinois at Urbana-Champaign (to appear in *Inf. Process. Lett.*).

J. Simon (1977), "On Feasible Numbers," *Proc. 9th ACM Symp. Theory Comput.*, pp. 195-207.

J. Simon (1981a), "Division in Idealized Unit Cost RAMs," *J. Comput. Syst. Sci.*, vol. 22, pp. 421-441.

J. Simon (1981b), "On Tape-Bounded Probabilistic Turing Machine Acceptors," *Theor. Comput. Sci.*, vol. 16, pp. 75-91.

C. Slot and P. van Emde Boas (1988), "The Problem of Space Invariance for Sequential Machines," *Info. and Comput.*, vol. 77, pp. 93-122.

M. Snir (1985), "On Parallel Searching," *SIAM J. Comput.*, vol. 14, no. 3, pp. 688-708.

L. Stockmeyer (1976), "Arithmetic Versus Boolean Operations in Idealized Register Machines," IBM Research Report RC 5954.

L. Stockmeyer and U. Vishkin (1984), "Simulation of Parallel Random Access Machines by Circuits," *SIAM J. Comput.*, vol. 13, no. 2, pp. 409-422.

C. D. Thompson and H. T. Kung (1977), "Sorting on a Mesh-Connected Parallel Computer," *Commun. ACM*, vol. 20, no. 4, pp. 263-271.

J. L. Trahan, M. C. Loui, and V. Ramachandran (1988), "Multiplication, Division, and Shift Instructions in Parallel Random Access Machines," *Proc. 22nd Conf. Inf. Sci. Syst.*, pp. 126-130.

U. Vishkin (1983a), "Implementation of Simultaneous Memory Address Access in Models That Forbid It," *J. Algorithms*, vol. 4, pp. 45-50.

U Vishkin (1983b), "Synchronous Parallel Computation: A Survey," TR-71, Dept. of Computer Science, Courant Institute, NYU.

D. J A. Welsh (1983), "Randomised Algorithms," *Discrete Appl. Math.*, vol. 5, pp. 133-145.

J. Wiedermann (1983), "Deterministic and Nondeterministic Simulation of the RAM by the Turing Machine," *Information Processing 83*, ed. R. E. A. Mason, Elsevier Science Publishers B. V. (North-Holland), New York, pp. 163-168.

Vita

Jerry Trahan was born [redacted] He received his B.S. in Electrical Engineering from Louisiana State University in December 1983 and his M.S. in Electrical Engineering from the University of Illinois in January 1986.